INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700  800/521-0600
Simulation of shared memory parallel systems

Mukherjee, Rajat, M.S.

Rice University, 1990
RICE UNIVERSITY

Simulation of Shared Memory Parallel Systems

by

Rajat Mukherjee

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

[Signatures and names of committee members]

Houston, Texas
April, 1990
Simulation of Shared Memory Parallel Systems

Rajat Mukherjee

Abstract

This thesis describes a method to simulate parallel programs written for shared memory multiprocessors. We have extended execution-driven simulation to facilitate the simulation of shared memory. We have developed a shared memory profiler, which, at compile-time, inserts simulation support code into the assembly code of the program to be able to extract the data address references at run-time. From the data address, we determine the nature of the reference, simulate the access and account for it. Programs to be simulated are written using Presto, an object-oriented parallel programming environment for shared memory multiprocessors based on C++. To validate the accuracy of our simulation methods, we have developed and evaluated an architecture model for the BBN Butterfly shared memory multiprocessor. The results of these tests are presented and discussed. We also describe extensions that would allow the simulation of shared memory systems with caches using execution-driven simulation techniques.
To my parents,

Runa and Rajen Mukherjee,

and my sister Rumki
Acknowledgments

Few human endeavors reach successful culmination without inspiration, moral support or aid at some stage from the people around.

When the issues are understood, but the ideas are only germinating, moral support is essential. I thank Dr. John Bennett for providing this support in more ways than one. A discussion with him always cleared up the fog that hung around the ideas.

I thank Dr. Bart Sinclair and Dr. Robert Jump for helping me out with the many doubts that I had at various points of the implementation of this project. I thank Dr. Willy Zwaenepoel for agreeing to serve on my thesis committee.

The person who contributed the most in helping me make this thesis a reality is Sandhya Dwarkadas, and I thank her for her constant help and advice. I also thank Bill Foundoulis and Sandhya, my officemates, for bearing with my being perennially logged on first to mu9, and later, to omicron2, our unfailing slaves (soon to be unceremoniously rechristened). Anand Dabak supplied the much needed stimuli when the going got slow or “bug”ging and my roommate Vijay helped me ruminate over my lunches and dinners by suggesting viable ideas from time to time.

I also thank John Mellor-Crummey, who arranged for me to use the GP1000 at Duke University, John Carter, who refrained from breaking my legs on the soccer field and helped me install C++ on the Butterfly, and Richard P. LaRowe at Duke, without whose help I would not have been able to get anything running on the Butterfly, leave alone Presto.
Contents

Abstract ......................................................... iii
Acknowledgments ............................................... iv
List of Illustrations ........................................... viii

1 Introduction ................................................... 1

2 Shared Memory Parallel Systems ......................... 3
   2.1 Shared Memory Vs. Message Passing .................. 3
   2.2 Shared Memory Architectures ......................... 7
       2.2.1 Variations in Shared Memory ..................... 7
       2.2.2 Caches in Shared Memory Architectures ......... 10

3 Simulation .................................................... 13
   3.1 Simulation .............................................. 13
   3.2 The Rice Parallel Processing Testbed ................. 18
   3.3 Limitations of the RPPT ............................... 21
       3.3.1 Language ........................................... 21
       3.3.2 Profiler ........................................... 23

4 Execution-Driven Shared Memory Simulation ............ 24
   4.1 The Basic Block Profiler ............................... 25
   4.2 The Shared Memory Profiler ......................... 26
       4.2.1 Generation of assembly file ..................... 27
       4.2.2 Assembly File Parser and Profiler .............. 28
4.3 The Execution of Profiled Code .............................................. 31
  4.3.1 Compilation, Linking and Loading ................................... 31
  4.3.2 Dynamically Generated Addresses ................................... 32
4.4 Simulation of Shared Memory ............................................... 33
  4.4.1 Determining Shared References ...................................... 33
  4.4.2 Simulating Shared References ........................................ 34

5 Simulation of the BBN Butterfly .............................................. 38
  5.1 The BBN Butterfly ......................................................... 38
    5.1.1 Switching Networks .................................................. 40
    5.1.2 The Butterfly Switching Network .................................. 42
  5.2 Architecture Model of the Butterfly ................................... 44
    5.2.1 Architecture Module ................................................ 44
    5.2.2 Butterfly Architecture Simulation Methodology .................. 45
  5.3 Assumptions Made in the Architecture Model ......................... 47

6 Validation and Results ....................................................... 49
  6.1 Algorithms ...................................................................... 49
    6.1.1 Subgraph Isomorphism ................................................. 49
    6.1.2 Gaussian Elimination .................................................. 52
    6.1.3 The Fast Fourier Transform Algorithm ............................ 54
  6.2 Results ........................................................................... 58
    6.2.1 Validation of the Shared Memory Profiler on the Sun .......... 59
    6.2.2 Overhead of Profiling Presto Code .................................. 64
    6.2.3 Overhead of Shared Memory Profiling ............................... 67
    6.2.4 Validation of the Shared Profiler on the Butterfly ............. 72
    6.2.5 Validation of the Butterfly Architecture Model ................ 79
  6.3 Overview of results ....................................................... 86
7 Evaluation

7.1 Implications of the Results ............................................ 88
7.2 Limitations ............................................................... 89
7.3 Possible Enhancements .................................................. 91

8 Conclusions

8.1 Conclusions ............................................................... 92
8.2 Future Work ............................................................. 94

Bibliography ................................................................. 95
6.1 Example graphs and K-formulas ........................................... 51
6.2 Subgraph Isomorphism - program structure ............................ 52
6.3 Execution and Simulated Times - FFT .................................... 60
6.4 Execution and Simulated Times - FFT .................................... 61
6.5 Percentage error in Simulation - FFT ..................................... 61
6.6 Execution and Simulated Times - ISO .................................... 62
6.7 Execution and Simulated Times - ISO .................................... 63
6.8 Percentage error in Simulation - ISO ..................................... 63
6.9 Overhead of Profiling Presto Code ....................................... 65
6.10 Simulation Timings - Presto Profiling Overhead ....................... 66
6.11 Remote References - Presto Profiling Overhead ...................... 66
6.12 Worst Case Simulation Overhead - FFT .................................. 68
6.13 Worst case simulation overhead - ISO ................................. 68
6.14 Simulation Overhead - FFT ............................................. 69
6.15 Simulated Execution Times - FFT ....................................... 69
6.16 Simulation Overhead - ISO ............................................. 70
6.17 Simulated Execution Times - ISO ....................................... 70
6.18 Timing Information with Local Data - FFT ............................. 73
6.19 Execution (Butterfly) and Simulated (Sun) Times - FFT ........... 74
6.20 Percentage Error in Simulation - FFT ................................... 74
6.21 Timing Information with Local Data - ISO ............................... 75
6.22 Execution (Butterfly) and Simulated (Sun) Times - ISO .......... 76
6.23 Percentage Error in Simulation - ISO ................................... 76
6.24 Timing Information with Local Data - GAUSS .......................... 77
6.25 Execution (Butterfly) and Simulated (Sun) Times - GAUSS ....... 78
6.26 Percentage Error in Simulation - GAUSS ............................... 78
6.27 Timing Information with Remote Data - FFT ........................... 80
6.28 Execution (Butterfly) and Simulated (Sun) Times - FFT ......... 81
6.29 Percentage Error in Simulation - FFT .......... 81
6.30 Timing Information with Remote Data - ISO .... 82
6.31 Execution (Butterfly) and Simulated (Sun) Times - ISO ... 83
6.32 Percentage Error in Simulation - ISO .......... 83
6.33 Timing Information with Remote Data - GAUSS .... 84
6.34 Execution (Butterfly) and Simulated (Sun) Times - GAUSS 85
6.35 Percentage Error in Simulation - GAUSS .......... 85
Chapter 1

Introduction

This thesis develops a method for the simulation of parallel programs written for shared memory parallel systems. The method used is a variant of execution-driven simulation [11, 13]. The simulation is carried out by using the Rice Parallel Processing Testbed (RPPT), which is a performance evaluation tool for the simulation of parallel computer systems.

In order to be able to analyze parallel programs executing on available parallel architectures and also to be able to design and evaluate new architectures, we need to be able to simulate the execution of parallel programs on parallel computer architectures. Before building an actual large scale multiprocessor system, it is wise to discover some of the bottlenecks or important flaws in the architecture by simulation.

The Rice Parallel Processing Testbed (RPPT) is a parallel simulation tool that uses a technique called execution-driven simulation [13]. However, the RPPT can only simulate message passing architectures. Shared memory multiprocessors are a very important class of architectures in part due to the attractiveness of the shared memory programming paradigm to programmers. This thesis describes an extension of the RPPT that makes possible the simulation of shared memory architectures. The approach used retains the advantages of execution-driven simulation.

The remainder of the thesis is organized as follows: Chapter 2 gives a general overview of shared memory parallel computing and discusses variations of the basic shared memory model. Chapter 3 begins with an overview of the methods used in
simulation. We then explain execution-driven simulation, and how it is incorporated into the Rice Parallel Processing Testbed. We also discuss some of the limitations of the RPPT with respect to the simulation of shared memory. In Chapter 4, we present our method used for simulating shared memory parallel systems that is based on a variant of basic block profiling. This method generates program timing information at compile-time, and also inserts code at compile-time that is used in the extraction of the data addresses referenced at run-time. We also explain how shared references are recognized and simulated. In Chapter 5, we provide an overview of the architecture of the BBN Butterfly, a shared memory multiprocessor based on a multistage interconnection network. The accuracy of our simulation techniques will be validated on the Butterfly architecture. In Chapter 5, we also describe the structure of the Butterfly switch that connects the processors. This switch determines the latency of the remote references and characterizes the performance of the Butterfly. In Chapter 5, we also explain the architecture model of the Butterfly that we use. Chapter 6 gives an overview of the algorithms used in the simulations. We describe our validation methodology and present an analysis of our results. We evaluate our simulation technique in Chapter 7 and discuss some of the limitations of our approach. In Chapter 8, we report our conclusions, and describe avenues for future work.
Chapter 2

Shared Memory Parallel Systems

In the last few years, there has been a significant increase in the interest and availability of parallel processors. Some of the motivations for the development of parallel computers include:

- *performance*: obtaining increasing power as more processors are available.

- *graceful growth*: allowing a system the ability to provide a range of power, thus allowing users incremental expansion of the system.

While it is simple to say that a parallel computer is one which has more than one processor, it is not so simple to categorize a particular parallel machine. Also, while it is easy to think of many processors working on a single problem, it is not trivial to actually implement a parallel algorithm on a particular parallel computer. Parallel algorithms may involve complicated communication patterns or synchronization. To efficiently parallelize a program, it is necessary to look into problems such as load balancing, deadlock, etc.

In the rest of this chapter, we present a brief overview of shared memory architectures, an important class of parallel computer systems.

2.1 Shared Memory Vs. Message Passing

As data communications technology advances progressively, the distinction between parallel and distributed processing becomes smaller and smaller. For instance, local
area networks of many powerful workstations connected together by communication lines can also be considered as a form of parallel computer.

In this thesis, multiprocessors are classified into two basic categories: multiprocessors using message passing and multiprocessors using shared memory. Figure 2.1 and Figure 2.2 emphasize the difference between shared storage and message passing organizations by representing the extreme cases. Actual machines may not follow one model or the other absolutely. Various hybrid organizations are possible.

There are many ways of classifying parallel machines, but the distinction between shared memory machines and message passing systems is a strong one because they represent two different programming paradigms. In shared memory systems, different subtasks access the same shared data structures, which reside in a single global address space. A pure message passing system does not have any shared global data. In message passing systems, the subtasks, which execute in different address spaces, communicate with each other by means of messages. Message passing parallel programs use primitives such as send or receive explicitly through channels, ports or mailboxes. Although these primitives often provide implicit synchronization, all required data motion must be explicitly specified.

There are two basic types of message passing: synchronous and asynchronous. In asynchronous message passing, the processes do not block after sending a message. Synchronous message passing involves blocking of processes. A process that sends a message cannot proceed until it has received some sort of acknowledgement from the receiver process.

In message passing, it is difficult to pass pointers as parameters to procedures because different processes have different address spaces. In order to pass a complex data structure between two processes executing on two different processors in a distributed system, it would be necessary to pack the data and send it as a message across the network.
Figure 2.1: Shared Storage Multiprocessor

Figure 2.2: Message Passing Multiprocessor
In contrast, a shared memory multiprocessor has no difficulty passing pointers because processors share a single address space. Therefore, there is no need to pack and unpack the data structures containing pointers in messages. Passing a list data structure simply requires passing a pointer.

Another problem with message passing systems is the difficulty of process migration because there are multiple address spaces. When a process is being migrated, all the operating system resources allocated to the process have to be moved together. The code and the stack of the process have to be moved because there is no easy way to translate the contents of different address spaces on the fly. In a shared memory multiprocessor system, a process migration only requires moving a process from the ready queue on the source processor to the ready queue on the destination processor because process control block, code, and stack are all in the same address space.

One important difference between the shared memory and distributed memory models is that the message passing model can easily be implemented on top of shared memory; all that is needed is a few shared data structures. The reverse is not true, however. Although several efforts have been made to implement shared memory on distributed systems [3, 7, 20, 21, 26], none of these has proven to be widely accepted.

The most important advantage of shared memory over message passing is that shared memory provides the programmer the simplest model for parallel programming - a single level of globally accessible memory. Programming shared memory systems is not very much different from programming ordinary sequential uniprocessors. The programmer just needs to monitor access to shared code and shared data structures.

While providing a simple parallel programming model, shared memory can also be a bottleneck in the system. When there are many processors trying to access common addresses, there is the possibility of contention, especially if all the processors and memory modules are on a single bus. Various enhancements have been attempted to make the shared memory model a viable one, e.g., addition of local caches, switching networks, etc. The next section gives an overview of some of these schemes.
2.2 Shared Memory Architectures

2.2.1 Variations in Shared Memory

The basic model for parallel shared memory computing is shown in Figure 2.3. This provides the simplest model for parallel programming. However, the memory can easily become the system bottleneck, since there is contention for the single shared resource.

One modification to this basic architecture is to replace the single shared memory module with a number of memory modules, all uniformly accessible by all the processors. This would imply that different modules could be accessed simultaneously by different processors, thus reducing memory contention and increasing system throughput. Figure 2.4 represents such an architecture. However, in such a system, the system bus, to which all the processing elements and memory modules are connected, becomes the system bottleneck very quickly.

To alleviate this problem, the bus can be replaced by a multistage interconnection network. An example of such a system is the C.mmp, constructed at Carnegie-Mellon University [23]. This is a classical multiprocessor designed to support up to 16 processors sharing 16 memory modules through a crossbar switch. Figure 2.5 is a representation of such a system.

The cost of accesses to any part of memory is uniform across all the processors; there can be 16 memory references simultaneously, provided that each reference is to a different memory port.

However, since the crossbar switching arrangement of C.mmp increases in size as \( N^2 \), where \( N \) is the number of memories and processors, it is not feasible to expand \( N \) to large values. In order to reduce the number of switches, various multistage alternatives, such as the Omega and Banyan networks, have been suggested.
Figure 2.3: Basic Shared Memory Model

Figure 2.4: Shared Memory Modules
Figure 2.5: Cross Point Switching as in C.mmp

Figure 2.6: Hierarchical Structure as in Cm*
Another way to avoid the single bus overload problem in shared memory multiprocessors has been implemented in the Cm*, also designed at Carnegie-Mellon University [18]. Cm* is a hierarchical system, as shown in Figure 2.6. The basic block is a processor-memory pair called a computer module (Cm). The memory local to a processor is also the shared memory in the system. Up to 14 Cm's are connected into a cluster. The cluster processors share a single bus. Clusters are connected via intercluster buses. The assumption of program locality is inherent in the Cm* organization. Efficient use of the system depends on ensuring that most of the code and data references made by a processor are held locally to that processor. Applications with close to linear speedup have been run on this architecture.

The assumption that only a fraction of the memory accesses made is actually to shared addresses has led to the development of local memories and cache memories for each processor. A model of computation based on local memories is shown in Figure 2.7. Each processor in this system has a local memory and all processors are connected to a global shared memory via an interconnection mechanism. All processors can access any location in the global shared memory, but cannot access the local memory of another processor. Such an organization would enhance the system efficiency by making fewer (only essential) references to the global common memory.

2.2.2 Caches in Shared Memory Architectures

Cache memories are small, high-speed buffer memories used in modern computer systems to hold temporarily those portions of the contents of main memory which are (believed to be) currently in use. Information located in cache memory can be accessed in much less time than that located in main memory. These caches only hold copies of the data that is being accessed by the processors, and are not distinct regions of the overall shared address space. Figure 2.8 depicts a multiprocessor system with a local cache at each processor.
Figure 2.7: Processors with Local Memories and Single Shared Memory

Figure 2.8: Processors with Single Shared Memory and Local Caches
The success of cache memories is the principle of locality. There are two aspects to locality: spatial and temporal. Spatial locality means that the loci of reference of the program in the near future are likely to be near the current loci of reference; related data items are like to be stored together, and instructions are mostly executed sequentially. Temporal locality means that the information that will be used in the future is likely to be in use already; in loops, both data and instructions are reused.

When a computer system contains more than one processor, each with a separate address space, adding a cache to each processor is an obvious benefit; the cache would behave in the same way as in uniprocessor systems, reducing the mean memory access time. However, in a shared memory multiprocessor system, where the processors share the same address space, the design of the memory hierarchy is more complicated. If the system includes caches for each processor, we have a cache coherency or cache consistency problem. This occurs because there are many copies of a particular data item in the available caches, and extra work needs to be done in order to keep the system of caches coherent; i.e., all copies of a main memory location in multiple caches remain consistent when the contents of that memory location are modified.

The mechanism by which coherency is maintained is called the cache coherency protocol. Maintaining coherency entails taking special action when one processor writes to a block of data that exists in other caches. The data in the other caches, which is now stale, must either be invalidated or updated to the new value. Also, if a read miss occurs on a shared data item and the memory has not been updated with the most recent value (as would happen in a copy-back cache), that most recent value must be found and supplied to the cache in which the miss occurred.

There are many cache coherency protocols, and we do not describe them in this thesis. These protocols mainly differ in the way in which they determine whether a block is shared, how they find out where the copies of the block reside, and how they invalidate or update copies.
Chapter 3

Simulation

An important problem facing the designers and users of parallel processing systems today is utilizing the available parallelism effectively to realize significant performance improvement over sequential systems. There is a need for accurate and effective tools to evaluate the performance of parallel systems under realistic workloads. Cost-effective techniques that evaluate the performance of a parallel algorithm on a particular architecture must be developed. Due to the difficulty of developing analytical models for the behavior of large parallel systems, it is expected that performance evaluation techniques for parallel systems will depend heavily on simulation.

Simulation models are an important supplement to analytical models because we can make the simulation as detailed as we want. However, since simulations generally are complex programs that will execute for a large number of iterations, they are expensive to run on a computer system.

In this chapter, we present the methodology of simulation of a parallel program on a parallel computer architecture. Simulations are run on top of the Rice Parallel Processing Testbed [13]. The parallel program is specified using Presto, an object-oriented programming environment developed at the University of Washington [4].

3.1 Simulation

Three traditional approaches to modeling the workload of a computing system are *distribution-driven simulation*, *trace-driven simulation* and *instruction-driven simulation*. 
lation [2, 6]. Distribution-driven simulations use a statistical model of the program to drive the simulation. In this approach a random process models the generation of data that move between the modules of a system during program execution. There is a statistical characterization of the way different data affect the resolution of conditional branches. The main advantage of this approach is that the simulation is very fast. The main disadvantage is the potential inaccuracy of the results.

Trace-driven simulations use a program's execution trace to drive a simulation model of an architecture. A trace is first obtained by actually executing the program on a computer that is similar to the one being simulated. As the program executes, the trace is generated by recording one or more distinguishing features of the execution (e.g., a list of the memory locations accessed). The second step is to use this trace as the input for the architecture being simulated. There are two major problems associated with this method. First, the simulated architecture must be very similar to that on which the trace was obtained. Second, traces derived from sequential systems are usually inappropriate for use in simulating a parallel system, since the order of execution is usually dependent on the order in which different parts of a parallel program complete execution. Since a sequential trace represents a fixed order of execution, it does not allow for possible overlap of different parts of the program.

In instruction-driven simulation, the execution of each instruction is usually simulated by the execution of several instructions on the simulation host computer. These instructions emulate the movement of data associated with the execution of an instruction. Instruction-driven simulation can be quite accurate, but can also be very slow, sometimes as much as 200 to 300 times slower than the actual execution of the parallel program [15].

Execution-driven simulation [11, 13] is a technique that speeds up the simulation without losing much of the accuracy of the results obtained. The simulations are driven by the actual execution of a program. This means that a real program is executed and directly drives a model of the computing system under study; i.e., the
parallel program is run on the host machine and timings are adjusted by calls to
architecture model routines so as to reflect the execution times and remote access
times of the target machine (the architecture being simulated). This means that
the execution of the program and the simulation of the architecture are interleaved.
Unlike instruction-driven simulation where the execution of each individual machine
language instruction is emulated by the execution of many host instructions (even
when the host and target machines are nearly identical), execution-driven simulation
simply executes the machine language instructions of the program directly on the
simulation host. Since it is not always possible to use a host processor that has the
same instruction set as the target processor, a technique called cross-profiling has been
developed. Cross-profiling attempts to match up the basic blocks on the host and the
target machines and changes the estimates of the basic blocks on the host machine to
reflect the estimates on the simulation target machine. Further details may be found
in [10, 12]. Cross-profiling is, however, not directly valid with the modifications made
to the profiler for shared memory profiling (see Chapter 7). Also, while instruction-
level simulation must update simulation time at least once during the emulation
of each instruction, execution-driven simulation will only update simulation time
when a process needs to communicate with another in some way (i.e., a remote
operation of some type is encountered). Such remote operations may include message
passing, synchronization, etc. Some of the differences between execution-driven and
instruction-driven simulation are illustrated in Figures 3.1 and 3.2.
Figure 3.1: Instruction-Driven Execution Cycle

Figure 3.2: Execution-Driven Execution Cycle
Figure 3.1 shows the basic cycle of an instruction-driven simulation. This cycle simulates the activity on a single processor, and each processor in the simulated parallel system will be simulated by a separate, but similar, cycle. The third block in the cycle, where the operation of an instruction is emulated, accounts for most of the large overhead associated with this kind of simulation, since several instructions must be executed by the host computer just to simulate the execution of a single instruction on the target processor. Note that the second block, where the time for the execution of the instruction is accounted for, must be executed once for each simulated instruction. This operation will also require the execution of several instructions to manipulate an event queue on the simulation host computer.

Figure 3.2 illustrates the basic execution-driven simulation cycle for one of the processors in a parallel system. This cycle is significantly faster than the corresponding cycle for an instruction-driven simulation of a parallel system, because it executes the instructions of the target processor instead of simulating their operation, and requires event queue operations only when a remote operation is encountered, rather than for each instruction. There is no overhead associated with an instruction that is not labeled, is not a branch instruction, or does not require a remote operation or synchronization.

The sequences of instructions executed in the large box of Figure 3.2 are called basic blocks. A basic block is a sequence of assembly language instructions that are executed in order exactly once every time the block is entered, and the sequence is not contained in any larger basic block.

A utility program called a profiler is used to modify the assembly code generated from the source program. Those instructions that branch or are labeled (start of each basic block) incur an overhead to increment a counter N. The profiler detects the basic blocks and for each block, it examines the instructions and uses the information about instruction and memory access timings to compute an estimate of the amount of time required to execute all of the instructions in the block. This estimate is
inserted into the source program in the form of a few instructions on each basic block boundary that increment the timing accumulator \( N \) by the amount of the estimate for that block.

Only when the simulated process encounters a remote operation is a significant overhead incurred. At that time, an event queue insertion must be processed and the user-supplied program that defines the simulation of the remote operation (message passing/remote data access) must be executed. The way elapsed time is accumulated between process interaction points in the variable \( N \) accounts for the high accuracy of execution-driven simulation.

Since we have incrementing instructions at all possible branch points, we are guaranteed that all the conditional branches of the program are timed during execution, so that all the data dependent properties of the program are taken into account during simulation. This gives rise to a very accurate representation of the system's workload, where all process interaction points occur in the proper sequence and at the proper simulation time. Also, the overhead required to simulate data movement within a processor is avoided.

For simulating shared memory, the basic block profiler described above is modified to generate the data addresses referenced at run time. The structure of the shared memory profiler is described in Chapter 4.

### 3.2 The Rice Parallel Processing Testbed

The Rice Parallel Programming Testbed (RPPT) is a performance evaluation tool for studying the execution of parallel programs on concurrent systems. It uses execution-driven simulation to accurately describe the data dependent flow of execution in a parallel program and to efficiently provide estimates of user code execution times. Detailed simulation models of the interconnection structure of the system being simulated account for delays in performing remote process synchronizations or remote data accesses. During the execution, the timing information from the architecture
model is combined with the estimates of instruction execution times to obtain accurate estimates of the overall execution times of the parallel program on the chosen architecture.

There are four basic components in the RPPT software system:

- **Concurrent C**: An extension of the C programming language obtained by providing a set of procedures that permit a user to create and control processes, and pass information between processes [22].

- **C Simulation Package (CSIM)**: A process-level, discrete-event simulator implemented by extending Concurrent C with a set of procedures for event queue manipulation and statistics collection [14].

- **Architecture Simulation Preprocessor (ASIMP)**: A software translator that modifies Concurrent C programs by inserting simulation primitives used to simulate architecture delays that are due to interprocess communication and synchronization between processes on different processor modules.

- **Timing Profiler (TPROF)**: A program analyzer used to estimate the time required to execute those segments of a Concurrent C program that execute sequentially on a single processor, and to modify the compiled version of that program by inserting simulation primitives that account for these delays during a simulation.

Figure 3.3 illustrates the relationship between the four basic RPPT components, the user-supplied program and architecture specification. First, the Concurrent C program is modified by ASIMP, which uses the assignment of procedures to processor modules and the definition of the architecture's interconnection network to insert code (calls to Architecture Simulation (ASIM) routines) that simulates all remote procedure calls and data accesses. This modified program is then compiled by the standard C compiler. The resulting assembly language program is processed by the
timing preprocessor TPROF, which examines the instructions in each basic block and inserts timing instructions to account for the time required to execute the block code. The actual accounting for the passage of simulation time occurs within the ASIM routines that are called from the user's program. These calls are made because the user's program has been modified by ASIMP to call the ASIM simulation routines. The modified assembly language program can now be used by CSIM to simulate the program's execution on the specified architecture.

![Diagram](image)

Figure 3.3: The Rice Parallel Processing Testbed
3.3 Limitations of the RPPT

3.3.1 Language

The Rice Parallel Processing Testbed uses Concurrent C as the specification language for parallel programs. Concurrent C is an extension of the C language. There are additional routines which implement both synchronous and asynchronous message passing. A parallel program written using Concurrent C uses these routines explicitly to realize message passing and process synchronization.

Concurrent C was designed as a language to specify parallel programs for a distributed environment. It is easy to write a message passing program in Concurrent C, but not easy to write shared memory programs, because there are only very low level shared memory synchronization primitives like the semaphore operations \( P() \) and \( V() \). Most users of shared memory systems prefer to use higher level constructs like monitors and condition variables. Another problem with Concurrent C is that it currently runs on a uniprocessor (Sun) and not on any available shared memory multiprocessor; in other words, Concurrent C is not a real language for shared memory parallel programming.

Since the RPPT can be used to simulate only those programs that are coded in Concurrent C, it is the responsibility of the user to code a shared memory algorithm in Concurrent C, using only the shared memory routines provided. This step is necessary even if a program is already available, and costs considerable time and effort. Moreover, such a program would only run on top of the RPPT as a pseudo-concurrent program, since Concurrent C is not available on any parallel machine (with the exception of the Sequent, where it is has been ported on top of Presto [24]).

One approach to the problem is to add some high level shared memory constructs to Concurrent C to make it possible to represent a larger variety of shared memory parallel programs. This does not, however, eliminate the necessity to re-code algo-
gorithms in Concurrent C, or make it any easier to run the programs on any available parallel machine.

The other solution is to build support for some other specification language on top of Concurrent C, since simulation requires the programs to be Concurrent C programs. Presto is an ideal contender for this, since it is written in C++, an object-oriented extension of C, and supports lightweight processes like Concurrent C. Presto allows the user to create threads and start these threads executing in some procedure, which could be an object operation or a global procedure. High level synchronization constructs like Monitors and Condition Variables are available and synchronization between parallel processes can be clearly defined. The Presto programmer can also exploit the C++ facets of inheritance between classes, operator overloading, etc. Presto is a widely used environment for shared memory programming, with many readily available programs.

We have adopted the second approach. We believe it is more appropriate to use real application programs written by different programmers to study the effects of different types of sharing. Therefore, Presto has been chosen in preference to Concurrent C as the specification language for simulating shared memory programs. Since C++ code is actually translated into normal C code, the profiling techniques described earlier can still be used in order to update code simulation times during simulation of a Presto program on the RPPT.

Thus, it has been necessary to layer Presto on top of Concurrent C in order to exploit the available simulation tools. The actual shared memory program remains unchanged, thereby allowing for the program code to be profiled in its entirety. There are some differences between the functionality of Presto and that of Concurrent C. For instance, while Concurrent C binds processes to their procedures at create time, Presto does this at start time. These differences have forced some small changes to Concurrent C and ASIM (see section 3.2), so that the Presto programs being simulated do not have to be changed.
The shared memory simulation methodology that we have developed in this thesis allows a user to directly simulate any available Presto program on the RPPT, provided that an architecture model of the shared memory machine being simulated is available. The user has the added option of creating a Presto thread on a particular processor, unlike native Presto, which resorts to a global scheduling policy with random placement on available processors. This feature is essential for placement of the threads on processors for accurate simulation, and requires only the addition of an extra parameter (Node Number) to the Presto thread\texttt{\_start}() call.

3.3.2 Profiler

The RPPT requires another extension to be able to simulate shared memory programs as opposed to message passing programs. The profiling technique (TPROF - Section 3.2) must be modified to generate all the data addresses referenced by the program during simulation. We have developed a different profiler, called the Shared Memory Profiler (SPROF), which is a variant of the cache profiler described in [17]. This program introduces timing information into the assembly code of the program at compile-time like TPROF, but also inserts code that generates the data addresses referenced by the program at run-time. A detailed description of shared memory profiling can be found in Chapter 4.
Chapter 4

Execution-Driven Shared Memory Simulation

The main goal of execution-driven simulation is to avoid the high overhead, while retaining the accuracy, associated with instruction-level simulation of computers. The resulting system is program-driven, but the overhead is significantly reduced by inserting profiling code containing timing information at compile-time. The timing information obtained at compile-time is used at run-time to update process execution times dynamically while avoiding a detailed emulation of each instruction's execution. This technique results in a simulation that can be used to produce performance predictions that are nearly as accurate as those obtained with instruction-level simulation at a fraction of its overhead.

In this chapter, we describe a utility program called the shared memory profiler which profiles assembly language code in order to dynamically extract the data addresses accessed by the program. This profiler extends the concept of execution-driven simulation to the simulation of shared memory systems and provides a technique that is space efficient since it does not require storage of the data addresses generated. This approach is potentially faster than conventional instruction-level simulators, since it does the instruction decoding necessary for timing analysis at compile-time (during profiling).
4.1 The Basic Block Profiler

The economy of performance of our approach can be attributed to the advantages of basic block profiling. The design is based on the basic block instruction count profiler developed at Bell Labs [29]. However, our approach differs in its intended usage. Instead of generating a dynamic instruction count mix, we generate an estimate of the execution time that the program would require on the target system.

A program, once represented as an assembly language file, is viewed as a collection of code segments called basic blocks. A basic block is a series of consecutive assembly language instructions that are delimited by labels and branches. Basic blocks have the property that once the block is entered during execution, all instructions in it will be executed exactly once. In practical terms, a basic block begins with:

1. the first instruction in a file,
2. the first instruction of a subroutine,
3. a labeled instruction, or
4. an instruction immediately following a branch.

Blocks end with:

1. the last instruction in a file,
2. the return of a subroutine,
3. an instruction immediately before a label, or
4. a branch.

Since all the instructions in a basic block are executed once if the block is entered, accounting for the entire basic block can be done at the beginning of the basic block [10]. This keeps the profiling overhead low without sacrificing much in terms of accuracy. The timing information generated during compile-time is used to dynamically
update the execution time of the program at a basic block by basic block level. Thus, simulation overhead is restricted to occur once at each basic block rather than at each instruction.

4.2 The Shared Memory Profiler

We have modified the basic block profiler described above in order to generate the data addresses along with the type of reference (read/write). We ignore instruction address references because we assume that all processors will have local copies of the code. Since we are simulating systems without external caches, we assume that instruction references do not add any overhead to execution times beyond that which is accounted for in the profiler's estimate of the execution time. If the system has external caches, we also need to extract all the instruction addresses to model the cache behavior.

Like the basic block profiler, our profiler (the shared memory profiler) does a compile-time analysis of the code in order to determine the execution time taken by the basic blocks. Code is inserted at the head of each basic block to actually update the execution time during simulation. The accuracy of the simulation could be enhanced by inserting timing code between every two remote accesses, rather than at the beginning of each basic block (see Chapter 7).

The effective address of an indirect reference (one whose address is contained in a register, for example) cannot be determined at compile-time. In order to determine such an address, code needs to be inserted before every such reference to extract the address at run-time. We do this for all data accesses. Thus, consistency is maintained, since all data addresses are extracted at run-time. For every data reference, we also insert a call to a subroutine where the actual data address is examined and the access to memory is simulated, based on the architecture of the target machine. This captures the execution behavior of the program at run-time, insofar as the behavior of the instructions can be inferred from a compile-time analysis.
For each program file, the following information is generated:

1. the number of clock cycles required for each basic block
2. data addresses accessed by the program, and
3. a value to indicate whether the access is a read or a write.

The shared memory profiler is a variant of the version developed for cache simulation [17]. The cache profiler, in addition to generating the data addresses and maintaining timing information, like the shared memory profiler, also generates the instruction addresses at compile-time. This is because instruction addresses are necessary for detailed simulation of caches to be able to determine cache performance metrics such as hit ratio, etc. The advantage of this is that the shared memory profiler can be extended to be able to simulate caches. Most shared memory systems have some external caches and this extension should be possible.

Figure 4.1 shows the steps necessary in order to profile a program and simulate a shared memory architecture. The main steps are described below.

4.2.1 Generation of assembly file

Profiling is done at the assembly language level, since the assembly code is easily modified in order to capture the run-time information required. The Presto code and the actual shared memory parallel program being simulated are written in C++ [28]. C++ is an object-oriented extension of C, with provisions for type checking, operator overloading and inheritance. The C++ preprocessor actually generates C code from the C++ source (Figure 4.1 - step 2). Compiling this C code gives us the assembly language representation of our application program (Figure 4.1 - steps 3, 4 and 5). The profiler is now used to process the assembly code, as described in the next subsection.
4.2.2 Assembly File Parser and Profiler

We have developed a one-pass profiler, which parses the input assembly program and determines those instructions that signify the start of a basic block. The profiler inserts preliminary code at the start of every function. The inserted profiling code also includes calls to a subroutine for every data access in the program, where the access is actually simulated. This routine is a part of the architectural specifications.
Steps 6 and 7 in Figure 4.1 represent the profiling step. After these steps, we have modified assembly code, with code inserted to generate timing information, data addresses and calls to the routine that simulates the data accesses. This assembly code is then assembled (Figure 4.1 - step 9) to produce the final object file used in the simulation.

Profiling analysis therefore requires the following steps:

1. The program is broken down into a set of basic blocks.

2. If a data reference is encountered, code is inserted in order to extract its address dynamically at run-time and to determine the nature of the access (read/write).

3. For every data access made in the program, we insert a call to a subroutine `Data_Access()` where the access is simulated. The simulation depends on the actual address, the nature of the access and the architecture being simulated.

The lexical analyzer is called by the parser in order to recognize tokens in the assembly language code and return their values. The parser is called by the basic block profiler whenever it recognizes a line of assembly code. The basic block profiler passes a pointer to the type of instruction that is being parsed, along with some control information, to the parser. The parser recognizes the number of operands and the types of addressing modes by calling the lexical analyzer in order to recognize the different tokens.

If the addressing mode is indirect or absolute, the parser inserts the appropriate instructions in the assembly language code in order to extract the address reference at run-time.

Figures 4.2 and 4.3 show a piece of 68020 assembly code before and after profiling.
Figure 4.2: Assembly Code Before Profiling

Figure 4.3: Profiled Assembly Code
4.3 The Execution of Profiled Code

4.3.1 Compilation, Linking and Loading

The shared memory profiler produces a modified assembly language listing of the original program. The modified listing is then be assembled and linked with the following modules:

- The Presto run-time library (also profiled)
- The Concurrent C library
- The Architecture Simulation library
- The Discrete-Event Simulator library
- The Architecture Model

The routine called for each data access (by the code inserted during profiling) is a part of the architecture model. This routine makes calls to other architecture routines that perform the simulation of the shared memory architecture under study.

In the specific case of the Unix linker and loader, the executable format consists of three logical segments:

1. the text segment, which starts at the beginning of the second page in virtual memory,
2. the data segment, which starts at the first segment boundary immediately following the text, and
3. the stack, which starts at the highest possible address in virtual memory, and grows downwards.

The addresses in the text segment are guaranteed to be identical to the original program's addresses since they are generated at compile-time. The stack addresses,
which are generated at run-time, are also unaffected, since the stack is always in the same state as the original program while it is executing its own code. The data addresses generated at run-time, on the other hand, will have a constant offset from the original addresses because of the additional instructions and data inserted by the profiler. The effect of the additional instruction segment space is minimized since the offset to the data is in integral multiples of the segment size (128 KBytes). Offset of the data space because of the increase in the size of the instruction segment is unlikely (it does not occur in the programs used because of the small size of their instruction spaces). Even if this does happen, the locality of the data references of the program, which is what we are actually trying to model, does not change. In practice, a large number of programs use dynamic allocation. For these programs, in order to ensure that there is no relative offset within the program's data, the profiling code is linked into the executable first, causing a constant offset to all data addresses, since the program data and the dynamically allocated data are now contiguous. If the profiling code is not linked in first, the data for the profiler will be allocated space after the program data and this space will be in between the program data addresses and the dynamically allocated data addresses.

4.3.2 Dynamically Generated Addresses

Instruction addresses are deterministic and do not change after having been linked into the final executable. These addresses can be generated statically but are ignored because they are assumed to be local to the processor executing the instructions and therefore do not contribute to any additional overhead. These instruction addresses would need to be generated at compile-time if caches need to be simulated. The current simulation is limited to the simulation of systems without caches.

Not all data addresses can be generated at compile-time. Addresses that are referenced indirectly either through a register or through memory cannot be generated statically. Similarly, any absolute address references cannot be generated statically,
even though they can be determined, since such references would be at variance with
the data addresses generated at run-time (offsets due to profiling). Thus, to ensure
consistency, all data addresses are generated dynamically by inserting code that deals
with the address generation into the program.

4.4 Simulation of Shared Memory

4.4.1 Determining Shared References

The method used to determine references to shared memory is as follows. The profiler
described above (variant of the cache profiler described in [17]) generates all the data
references. However, we need to be able to identify which of these references are
remote, and which are not. An access to a shared variable does not incur an overhead
unless the physical memory is remotely located.

On the Sun, since shared variables have been statically declared in the same way
as global variables in the program, space for these variables is allocated in the Data
Segment while the local variables (of the various threads) are allocated space on the
stack (Stack Segment). Given an address, it can easily be determined if the reference
is to the stack area or the data area. Any access to the stack segment is considered
to be an access to the non-shared section of memory. This is true in the case of the
Butterfly architecture. In general, if the stack is not local, and is known to reside in
remote memory, we have to simulate stack accesses as remote accesses. The decision
depends on the mapping of the address space of the target architecture, and can be
changed by the user depending on the architecture that is being simulated.

All space that is dynamically allocated at run-time is considered to be in the
shared or global part of memory, and accesses to these addresses are treated in the
same way as accesses to shared variables. Dynamic allocation at run time will be
in the same region of memory, irrespective of whether the space was allocated for
a shared or non-shared data structure. The new operator is used frequently in all
Presto programs to allocate space dynamically for both local and shared variables, and it would not be unfair to assume that most systems (shared or otherwise) would have only a single mode of allocating memory dynamically at run time. In fact, Presto on the Sequent uses a version of malloc called shmalloc (very similar to malloc) to make all dynamic memory allocations at run-time, not differentiating between allocation for shared variables and allocation for local variables.

We have the generated data addresses, but these are specific to the way space is allocated on the host machine (Sun). Since the Sun is a uniprocessor, the C++ compiler on the Sun preprocesses the shared declarations to make shared variables appear as ordinary global variables. We cannot obtain any meaningful information from the actual address value since the addresses generated are specific to the host architecture and the address itself has no significance on the target architecture. However, from the address, we can determine whether the access is to the stack segment or to the Data/BSS segment. Any access to the stack segment is a local access and does not incur any extra overhead. If the data reference is to the Data/BSS segment, we can say that the address is to a global location, and that the corresponding reference on the target machine would also be to shared memory, even though the actual address could be quite different.

4.4.2 Simulating Shared References

In the current version of the RPPT, a routine called UserSend() is called every time a message is passed from one process to another. Since message passing is explicitly coded into the user program, it is explicitly known when the calls to this routine are made.

Since we are trying to simulate a shared memory program, where there are no calls to any message passing routines, we have no explicit way of knowing when a shared variable is referenced within the program. Although the shared variables used in the program are declared explicitly as such, we still cannot determine at compile-time
when a shared location is accessed. This is because global addresses may be assigned to local pointer variables within the program, and these assignments do not indicate at compile-time whether the address that is finally generated is the address of a local or shared (global) variable.

Given a Presto program, we simulate the running of the program on the RPPT as follows: The profiler adds code that takes care of the timings of all the C code that results from the Presto program written in C++. Whenever an ASIM routine (such as AsimCreate, AsimFork, etc.) is called, the current process is delayed for an amount of (simulation) time equal to the time accumulated by the profiling step (time taken for the code to execute). Since no message passing routines are called from within the Presto program, there are no remote accesses (in the RPPT sense). However, some of the accesses to memory (data references) are to shared (remote) data while the remaining accesses are to private (local) data. These shared data references may actually have different access times, depending on the shared memory architecture we are simulating. Moreover, the accesses to shared memory may need to be mutually exclusive (for example, two processes cannot read and write the same location simultaneously). Thus, when a process makes a reference to shared memory, it may need to be delayed/blocked until further notice, etc., depending on the architecture of the shared memory machine that is being simulated. The behavior of the process upon making a shared memory request is what characterizes the architecture. Just as the procedure UserSend() describes a message passing architecture, a procedure DataAccess() is called by processes when they make references to shared memory, and will represent the architecture of a shared memory machine being simulated.

From the data addresses extracted by the profiling step, we can determine whether a particular memory access is to local memory or to shared memory. However, we must also be able to tell where the referenced data is located with respect to the overall layout of global memory on the target architecture. The entire shared address
space can be organized physically as local and remote modules, and given the generated address, we should be able to say whether a particular access to shared memory was to a local memory module or to a remote one. For example, if the shared memory of the target architecture is organized as a hierarchical memory system, accesses to different remote modules would incur different overheads.

In order to be able to detect this difference, we must partition the global space on the host machine into a number of parts (as many as the number of physically distinct memory modules on the target processor) and associate each such partition with one (or more, depending on the target processor-memory layout) physical node of the target machine. This partitioning must be done based on some information about data distribution in the target memory system for the application under consideration. We expect that in shared memory systems where the placement of data can enhance the performance of the application, the user has some control over the placement of the shared/global data, either explicitly or by giving hints to the compiler. It is this initial placement of the data that we should try to model by appropriately partitioning the host's global address space over all the logical nodes (which represent the target machine's physical processors) and assigning the partitions correctly to these nodes. By using different partitioning schemes, we can also investigate how the efficiency of execution changes with the distribution of data in the memory of the target machine.

The actual assignment of the partitions of the global address space to the processors of the architecture being simulated can be accomplished by means of the Address Mapping primitives of Concurrent C. These are:

- **AssociateAddr**\((\text{start\_address, end\_address, logical\_node})\)

  ```
  char  *start_address;
  char  *end_address;
  int   logical_node;
  ```
• `logical_node = LookupAddr(address)`
  ```
  char *address;
  ```

These primitives manipulate a data base of address ranges and their associated logical nodes. Their primary use is in modeling partitioning of data among the processors in a multiprocessor system. `AssociateAddr()` enters into the data base the address range which starts at `start_address` and ends at `end_address` (both inclusive) and associates with that range the logical node `logical_node`. Address ranges are assumed to be non-overlapping. `LookupAddr()` returns the logical node associated with a particular address if there is one, and returns a negative number if there is none. Once a memory reference is determined to be to a shared location in the target memory system, the actual logical node associated with that particular memory location can easily be obtained by looking up the address in the data base that has been initialized in the architecture initialization routine, reflecting a feasible distribution of the data on the target memory system. Once we ascertain whether the access is to a local memory module or a remote one, the overhead incurred in making the access can be determined, and the corresponding access can be suitably simulated by the architecture model of the target machine.
Chapter 5

Simulation of the BBN Butterfly

In order to be able to validate the accuracy of our simulation method, we developed an architecture model for the BBN Butterfly, a shared memory multiprocessor based on a multistage interconnection network. In this chapter, we first describe the architecture of the Butterfly and then describe the architecture model of the Butterfly that we use in our simulation of the machine. We describe the simulation methodology and outline the assumptions that have been made in the architecture model of the Butterfly.

5.1 The BBN Butterfly

The BBN Butterfly (GP1000) multiprocessor is a commercial, modular computer system composed of multiple processors and memory modules connected by a high performance network-interconnect system, the Butterfly switch. Each processor node consists of a Motorola 68020 microprocessor, an MC 68881 floating point coprocessor, a Processor Node Controller (PNC) and 4 Megabytes of random access memory. In configurations containing as many as 256 processor nodes, the Butterfly offers up to 1 GByte of shared memory and 600 MIPS of aggregate processing power. The block structure of the Butterfly architecture is illustrated in Fig 5.1.

The PNC (a 16-bit processor) intercepts all memory references from the microprocessor and accesses either local or non-local memory on its behalf. The PNC also handles all incoming memory requests from non-local processors, arriving via the Butterfly Switch.
Figure 5.1: Architecture of the BBN Butterfly

The Butterfly switch implements remote memory accesses by using packet switching. Programmers need not make special allowance for remote memory accesses, so the total combined memory of all the processor nodes appears simply as one large, shared memory space. There is support for atomic operations to ensure that multiple threads of computation can synchronize their access to shared data.

Special circuitry in each processor node automatically directs memory references to either local or remote memory. The switching network performs 32-bit remote reads in about 7 microseconds and remote writes in about 5 microseconds (256 node Butterfly). The reason for the difference in the remote read and remote write times is that a processor needs to wait for the read information to come back across the network (remote read) but is released by the PNC after the write message is sent off (remote write). Therefore, a remote read involves two passes across the Butterfly network while a remote write only requires one pass. Local accesses are directly handled by the local memory on the node and incur additional overhead only when the local bus is being used for a request from a remote processor. Each path through
the switch supports interprocessor data transfer at 32 MBit/sec. If a pathway is inoperable, alternate paths can be configured and will be used automatically by the system if specified.

From an architect's viewpoint, the most interesting features of the Butterfly machine are the Butterfly Switch and the PNC, as together, these components define the time penalty associated with non-local memory accesses and hence the intrinsic machine latency. However, because there is a time differential between local and non-local memory references, and because messages passing through the switch can interfere with one another, the way in which data are distributed throughout the shared address space also plays an important role in determining the machine's overall performance.

### 5.1.1 Switching Networks

Three interconnection architectures for processors dominate parallel processing: buses, hypercubes and multistage interconnection networks. In a bus-based system, the processors, memory modules and input/output devices are connected to one or more high speed buses. All memory modules are equally accessible to each processor, making the shared memory model a natural for programming. The bandwidth of the common bus limits the scalability of such an architecture.

In a hypercube architecture, each processor has its own memory and is connected to a number of other processors. The number of connections is called the dimension of the hypercube. Hypercubes can be expanded to a large number of processors since there is no common bus. However, when a message is sent between two processors that are not directly connected, it must be forwarded through intermediary processors.

A machine based on a multistage interconnection network connects processors and memory modules through a specialized switching network. The entire memory can be accessed directly by any processor, and the system is scalable since the switching network expands and the switching bandwidth increases as processors are added.
Many processors can simultaneously access many memories because multiple paths exist through the network.

One simple example of a switching network is the crossbar switch. The rectangular grid of \( n \) buses from the \( n \) processors and the \( m \) buses from the \( m \) memories, as shown in Fig 5.2, forms the crossbar. Crossbar switches allow all processors to access memory simultaneously, in parallel, as long as each processor reads from, or writes to, a different module. Crossbar switches have no upper limit on throughput. Each additional processor or memory module adds a new parallel path through the switch. The interconnect bandwidth grows linearly with the number of processors. The number of switch nodes grows as \( n \times m \). Since most of the complexity is concentrated in the switch nodes, the cost would also increase along the same scale.

![Crossbar Switching Network](image)

Figure 5.2: Crossbar Switching Network

A different kind of processor and memory interconnect reduces the complexity of a crossbar switch by linking multiple crossbars as nodes in a tree-like network. The Butterfly network (Figure 5.3) is one such.
5.1.2 The Butterfly Switching Network

The Butterfly switch network is a multistage network made of 4-by-4 crossbars to connect processors and memory modules in levels. Fig 5.3 represents the Butterfly switch for connecting 16 processors and 16 memory modules in 2 levels. Like the crossbar, the Butterfly switch network is self-routing and serial in nature. Unlike a crossbar switch, whose cost grows as $n^2$, multistage networks like the Butterfly switch attach $n$ processors to $n$ memories at a cost that grows only as $n \log n$. The factor $\log n$, which corresponds to the number of levels in the tree, is the number of stages;
the logarithm base, which corresponds to the tree fan out, is the number of inputs (and outputs) to each switch node.

The maximum parallelism in access to memory through a multistage network is as good as that of a crossbar switch [27]. All processors can access memory simultaneously and in parallel, provided that no two processors try to take the same output from a particular node (i.e. same branch of the tree). This is a stronger condition than the crossbar switch requirement that each processor access a different module in parallel, because even if two processors access different memory modules on a Butterfly switching network, it is possible that they contend for the same output of a switch. Hence, contention in multistage networks is potentially a problem. However, this problem can be alleviated by adding some extra switch nodes to introduce alternate paths, which can be used in the case of contention at a switch.

The size of the switch-node base is another important design decision that greatly reduces the potential for contention. Switches with larger bases require fewer columns and therefore fewer switch nodes to implement a switch of a given size. Since the number of nodes that a message must pass through directly affects the probability that messages will conflict as they travel through the switch, the switch with a large base will have a lower conflict rate. The larger the switch base, the closer the network is to a full crossbar. Therefore, we have to make a tradeoff between the size of the switch node base (a larger base reduces the probability of collisions) and the switch node cost and complexity (greater cost for larger base).

As messages pass through the switch, routing of the messages can be done 'on the fly' since the ports are uniquely labeled. Each message incorporates a header containing the label of its destination port. The insertion of the messages into the switch from different switches occurs asynchronously, and the time taken for a message to propagate from its source to its destination depends on the number of inputs to the switch and the loading on the switch during routing.
The network messages, containing non-memory memory addresses, a data field and some control bits, are approximately 80 bits long and, to keep the complexity (measured here in terms of inter-stage wiring) of the switch nodes within manageable limits, the switch nodes serialize the packets into 8-bit chunks. These chunks are piped through the switch at a rate of one chunk every 100 nanoseconds, resulting in a minimum switch latency of approximately 10 cycles per packet (1 μsec at each switch node) [19].

5.2 Architecture Model of the Butterfly

5.2.1 Architecture Module

All the code responsible for the simulation of the shared memory architecture has been developed as an architecture module and placed in the architecture library of the RPPT. The most important components of the shared memory architecture model of the BBN Butterfly are the following:

1. The routine ArchInit(), which initializes the Butterfly network of the specified size, starts off the Network Simulation Process, and initializes the data structures used for the simulation and statistics collection.

2. The routine ArchFinish(), which is called at the end of the simulation to clean up and print the statistics that have been requested. For example, these statistics might include the overall simulated time, the per-node utilizations, idle times, number of collisions, average remote access times, and the number of remote accesses made over the Butterfly network.

3. The routine Remote_Access() that is called for every non-local data access made. This routine is called from within the routine Data_Access() which is called for every data access due to the profiling code inserted. The actual mechanism is described in the next section. The routine Remote_Access()
introduces request packets into the Butterfly network to be handled by the simulation process NetSim().

4. The routine NetSim(), which actually simulates the network by looking at the many switches, moving packets from stage to stage, and waking up the processes blocked on remote accesses when the data requested is available at the end of the network.

5. A routine Distribute(), which is responsible for distributing the data space among all the logical nodes by means of the use of the Concurrent C routines AssociateAddr() and LookupAddr() described in Chapter 4. This mapping determines whether a data address is to a local module or a remote one. Changing this distribution of data over the local spaces owned by the available nodes will reflect a different distribution of the program data on the target Butterfly architecture.

6. A routine Alloc.on.node() which, like Distribute(), is used to allocate data to processors using AssociateAddr(). This routine is called from the user program for mapping particular data structures to nodes.

The various routines described above, along with their supporting code are all compiled and linked together as a single architecture module. This module is then linked with the profiled program, the Concurrent C library, the Simulation Package and the profiled Presto library to produce the final simulation executable.

5.2.2 Butterfly Architecture Simulation Methodology

The program code and the Presto code are both profiled by the shared memory profiler as described in the earlier chapter. The profiler inserts timing information on a basic block by basic block level. The profiler also inserts calls to a routine Data.Access(). This routine is called for each data access.
The data space of the host machine is distributed among the number of Butterfly processors being simulated so as to reflect the chosen distribution of data on the target machine. This can be done by the routine Distribute(), and also from within the application program by calls to the routine Alloc_on_node(). Both these routines map a range of addresses to a processor via the Concurrent C routine AssociateAddr(). After this is done, each data address on the host machine is associated with a particular target processor.

When the routine Data_Access() is called, the data address and the nature of the access (read/write - generated by the profiler) are examined. From the data address value and the available mapping of the data to the processors, it can be ascertained whether the access is a local or a remote access. If local, there is no simulation overhead. If the access is remote, we account for the time that has been accumulated so far by basic block profiling. A request packet is then created and placed on the Butterfly network by the routine Remote_Access(), to be handled by the simulation process NetSim(). If the access is a read, the process that made the remote request blocks until it is awakened by the simulation process. If it is a write, it is released after a certain time interval, while the write request is placed on the network.

The simulation process is an infinitely looping segment of code, which examines the states of the 4x4 switches that constitute the Butterfly network, and moves the packets from stage to stage, delaying for the appropriate times in order to account for switch and arbitration delays. If a packet comes off the last stage of the Butterfly network, the simulation process examines the packet to determine if the packet is a request for data or if the packet is a reply packet.

If it is a read request, then a new packet is placed on the network by the simulation process. If it is a reply packet (reply to read), then the process that originally made the remote request is awakened by the simulation process. Figure 5.4 depicts the architecture simulation methodology pictorially.
5.3 Assumptions Made in the Architecture Model

There are some assumptions and approximations made in the architecture model. We assume that all local references, including data and instruction address references, do not incur any overhead. This is not very accurate, since the references made to the local memory from remote processors have a loading effect on the local memory latency. This effect may be accounted for by means of multiplying all the memory access times in the profiler tables by a constant that suitably represents the loading effects of external references to local memory. It is, however, an experimental exercise to arrive at a proper value for such a factor.

The Butterfly network, as simulated, does not provide any path redundancy. In the real system, there could be alternate paths available in the case of contention in the network. This could affect the performance figures, but there is no available data about the exact configuration of these extra switches. In the simulation, we use
buffers at each switch and the size of the buffers is the only parameter that we can alter besides the actual switch delay times and switch arbitration times.

Another discrepancy in the simulation model is that the network simulation is carried out in a synchronous way. In the real Butterfly network, a packet can arrive at a switch at any time; i.e., the switches operate independently and asynchronously. However, modeling this asynchronous behavior of the switches would entail using a separate process for modeling each switch in the network. This is not feasible while simulating a large scale network for reasons of space and simulation overhead. Therefore, we use a single process to simulate the Butterfly switching network, which moves packets synchronously in groups from switch to switch.

The results of the simulation depend directly on the distribution of the data among the target processors being simulated. Thus, the accuracy of the simulation model depends heavily on how close the selected mapping is to the mapping actually used on the Butterfly. It is the responsibility of the user to accurately reproduce the mapping of the data as it would be on the Butterfly. This is actually quite simple, since allocation of shared data on the Butterfly can easily be monitored and changed, and the corresponding data distribution in the simulation can also be achieved. The only drawback is that there is no way of specifying or changing the mapping of data to nodes automatically.
Chapter 6

Validation and Results

In this chapter we describe the methodology used to validate our technique for simulating shared memory systems. The BBN Butterfly was chosen as our target architecture. The Butterfly is a shared memory multiprocessor with 64 processor nodes connected by means of a multistage interconnection network. We will describe the functionality of the selected Presto programs and how each of these programs was partitioned for parallel execution.

The quantitative results of our execution-driven simulations, using an architecture model of the BBN Butterfly, are also presented, along with the data obtained from actual execution on the Butterfly.

6.1 Algorithms

In this section we will describe the algorithms used to validate our technique. These algorithms include Subgraph Isomorphism, Gaussian Elimination and the Fast Fourier Transform.

6.1.1 Subgraph Isomorphism

The Subgraph Isomorphism problem is an NP-hard problem, often used in artificial intelligence applications. The algorithm we use takes two graphs as input and determines all instances of the smaller graph in the larger graph.
The simplest solution to this problem is to enumerate all possible mappings of the nodes of the smaller graph onto those of the larger graph and then to test to see if each of these mappings is feasible. Exhaustive tree search generates an exponential number of tree-nodes, and is practical only for small graphs. If the graphs are large and/or dense, exhaustive search leads to significant wasted computation caused by the need to investigate every mapping, even though many of these comparisons are unproductive. Practical algorithms proceed in two phases: a refinement procedure followed by tree search. The algorithm that we use introduces refinement in the tree search procedure by using a method called \textit{resolution}[8]. Due to this refinement procedure, only a few of the nodes can survive in the later tree expansion. This method is described below.

We are given the representations of two directed graphs in the form of the corresponding adjacency matrices. We are required to find all the occurrences of the smaller graph in the larger one by inspecting the information contained in the adjacency matrices. The relations among the vertices (the constraints) can be embedded into the K-formulas for the nodes of the graphs [5]. K-Formulas show which nodes are connected to which other nodes. An example is shown in Fig 6.1.

Given the K-formulas, it is evident that each node in the smaller graph can be mapped to a node in the larger graph if that particular node has branches to a greater number of nodes (in the example, node 2 can be mapped to nodes a, b, c and e, but not to d.)

Thus, from the adjacency matrices (matrix representations of the graphs), we have to find the possible mappings, and then, as we map more and more nodes, we have to see if the mappings so far are still feasible. This leads to a tree structure that represents an exponential number of possible mappings. The resolution process that we use helps to prune this tree so that the algorithm runs in a shorter time than if we were to explore each branch of the tree completely.
First, we generate sets of mappings (called constraint sets) for each node of the smaller graph by examining the adjacency matrices of the two graphs. Once the constraint sets are generated (as matrices), we use them in the resolution process. Resolution starts with a matrix of all 1's (each node of the small graph possibly mapped to every node of the larger graph). Resolution continues by mapping one node in the small graph to one in the larger graph one at a time until all nodes have been resolved and checked. If a tree-node survives resolutions with all constraint sets (4 in this case), then the tree-node represents an isomorphism. The output is a list of all the possible mappings of the nodes of the smaller graph into the nodes of the
larger one, i.e., all the subgraphs of the larger graph that are isomorphic with the smaller one.

The shared memory version of this algorithm has one main thread that first reads the input graphs, generates the constraint sets and then starts a variable number of slave threads to do the resolution and write out the results [16]. We maintain a work stack, protected by a monitor, from which the slaves get and to which the slaves add work arrays. Fig 6.2 depicts the shared memory algorithm.

![Diagram](image)

**Figure 6.2: Subgraph Isomorphism - program structure**

### 6.1.2 Gaussian Elimination

Gaussian Elimination is a technique used for solving a system of linear equations that appears in many applications in engineering, finite element analysis, and mathematics. While Gaussian Elimination can be easily parallelized, the algorithm still contains
non-trivial synchronization constraints. The goal is to triangularize a given square matrix $M$ according to the algorithm that follows:

**Step 1.** Set the iteration index $j$ to 0.
**Step 2.** Repeat steps 3 to 7 until $j = \text{problem size}$.
**Step 3.** Determine the pivot row $p$ such that $\forall M(i,j), i \geq j : |M(p,j)|$ is maximum. If $|M(p,j)|$ is zero then the matrix is singular. Stop.
**Step 4.** If $p \neq j$ then exchange rows $p$ and $j$ (Pivoting).
**Step 5.** $\forall i > j$, let $M(j,i) = M(j,i)/M(j,j)$.
**Step 6.** $\forall i, k > j$, let $M(i,k) = M(i,k) - M(i,j) \times M(j,k)$.
**Step 7.** Increment $j$.

In the algorithm, **Step 5** and **Step 6** are usually referred to as the elementary row transformations. Pivoting (**Step 4**) is necessary to ensure the numerical stability of the algorithm and the accuracy of the final solution. Note that the synchronization implied by the pivoting step limits the parallelism.

**Shared-memory-based Solution**

In the parallel algorithm, the coefficient matrix is considered as an object and is shared among all threads participating in the solution [1]. The matrix is distributed among the threads such that an even distribution of the load is obtained. A central thread determines the next pivot row at each iteration, and also performs the test for singularity. After the pivot row is determined, the matrix is distributed among dynamically created threads which perform the elementary row transformations on their respective partitions in parallel. These threads are destroyed after finishing their computations. Before getting destroyed, each thread increments a global counter, which is protected by a monitor, to inform the central thread of its termination so that the central thread can determine when to start the next iteration.

There are a couple of alternatives in the algorithm which achieve different distributions of the load. In the first case, a separate thread is created for each column of the coefficient matrix. This is motivated by the simplicity of the design and the inexpensive process creation facility provided by the environment. In the second case, the
number of threads created in each iteration is limited to a fixed number of threads, reflecting the number of available processors in the system.

6.1.3 The Fast Fourier Transform Algorithm

The Discrete Fourier Transform (DFT), which is primarily used to study and analyze the frequency response of a system, is an important tool for digital signal processing.

Consider a sequence $x(n)$ of equally spaced samples obtained from a full period of a continuous periodic signal $x(t)$. The DFT is a Fourier representation of the sequence $x(n)$, and it corresponds to samples equally spaced in frequency of the continuous Fourier transform of the signal $x(t)$. The DFT $X(k)$ of an $N$-point sequence $x(n)$ is given by

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j(2\pi/N)nk} \quad (6.1)$$

Due to the nature of the complex exponential $e^{-j(2\pi/N)nk}$, which is periodic in $k$ with a period of $N$, $X(k)$ is also periodic; i.e., $X(0) = X(N)$, $X(1) = X(N+1)$, etc.

The inverse DFT can be obtained from the equation

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)e^{j(2\pi/N)nk} \quad (6.2)$$

For convenience in notation, these equations are generally written in terms of $W_N$, where

$$W_N = e^{-j(2\pi/N)} \quad (6.3)$$

Therefore, we will express the DFT equations as

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn} \quad (6.4)$$

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)W_N^{-kn} \quad (6.5)$$

The Fast Fourier Transform algorithm improves the efficiency of the computation of the DFT and its inverse by decomposing the computation into successively smaller
DFT computations. This process exploits the symmetry and periodicity of $W_N^{kn}$. In our application, we will consider only the case of sequences whose size is an integer power of 2; i.e., $N = 2^v$.

Since $N$ is even we can consider computing $X(k)$ by separating $x(n)$ in two $N/2$-point sequences, one consisting of the even-numbered and the other of the odd-numbered points of $x(n)$:

$$X(k) = \sum_{n \text{ even}} x(n)W_N^{kn} + \sum_{n \text{ odd}} x(n)W_N^{kn} \quad \quad \quad (6.6)$$

We can do a substitution of variables $n = 2r$ for $n$ even and $n = 2r + 1$ for $n$ odd:

$$X(k) = \sum_{r=0}^{(N/2)-1} x(2r)W_N^{2rk} + \sum_{r=0}^{(N/2)-1} x(2r + 1)W_N^{(2r+1)k}$$

$$= \sum_{r=0}^{(N/2)-1} x(2r)(W_N^{2})^{rk} + W_N^{k} \sum_{r=0}^{(N/2)-1} x(2r + 1)(W_N^{2})^{rk} \quad \quad \quad (6.7)$$

And using the equality

$$W_N^2 = e^{-j(2\pi/N)} = e^{-j(2\pi/(N/2))} = W_{N/2} \quad \quad \quad (6.8)$$

We can rewrite $X(k)$ as

$$X(k) = \sum_{r=0}^{(N/2)-1} x(2r)W_N^{rk} + W_N^{k} \sum_{r=0}^{(N/2)-1} x(2r + 1)W_N^{rk}$$

$$= \sum_{r=0}^{(N/2)-1} g(r)W_N^{rk} + W_N^{k} \sum_{r=0}^{(N/2)-1} h(r)W_N^{rk} \quad \quad \quad (6.9)$$

$$= G(k) + W_N^{k}H(k), \quad k = 0,1,\ldots,N-1$$

Each of the sums in Eq. (6.9) is an $N/2$-point DFT, and each can be computed by separating their corresponding sequences $g(r)$ and $h(r)$ into their even and odd components. It is important to observe that $g(r)$ and $h(r)$, as well as their transforms $G(k)$ and $H(k)$, are periodic, with a period of $N/2$. This process of separation into the even and odd components is repeated until the subsequences have only 2 elements.
The evaluation of the DFT for a 2-point sequence is straightforward. From Eq. (6.4),
\[ X(k) = x(0) + x(1)W_2^k, \quad k = 0, 1 \]
\[ \ldots \]
\[ X(0) = x(0) + x(1)W_2^0 = x(0) + x(1) \]  \hspace{1cm} (6.10)
\[ X(1) = x(0) + x(1)W_2^1 = x(0) - x(1) \]

With this result and Eq. (6.9) we can progressively compute the DFT for the sequences of 4, 8, \ldots, up to \( N \) points.

To implement the Fast Fourier Transform algorithm we need only one \( N \) element array to store the original sequence and all intermediate steps of the computation. This can be inferred from Eq. (6.9) which shows that only \( G(k) \) and \( H(k) \) are needed to evaluate \( X(k) \) and \( X(k + (N/2)) \), making it possible to reuse the locations in memory where \( G(k) \) and \( H(k) \) are stored. Thus, the first step in the algorithm is to rearrange the original sequence to obtain \( N/2 \) subsequences of length 2 in the original array. To do this, we first separate the original sequence into its even-numbered and odd-numbered samples; we store the even-numbered samples in the first \( N/2 \) positions and the odd-numbered in the last \( N/2 \) positions of the array. We check the least significant digit of the binary representation of the index, and if this bit is zero, we move the point to its corresponding position in the lower part of the array; if this bit is one, we move the point to its position in the higher part of the array. To separate these subsequences into their even-numbered and odd-numbered samples, we check the second least significant digit of the binary representation of the index and move the point accordingly. When this process completes, a point originally located in position \( (b_{\log N-1}b_{\log N-2}\ldots b_1b_0) \) will have traded places with the point located at \( (b_0b_1\ldots b_{\log N-2}b_{\log N-1}) \).

The time required to compute the DFT using Eq. (6.1) is \( O(N^2) \): we need \( O(N) \) complex operations for each one of the \( N \) points of \( X(k) \). The Fast Fourier Transform algorithm requires \( O(N \log N) \) operations: \( \log N \) steps with \( O(N) \) complex operations in each step.
The algorithm to compute the Fast Fourier Transform for an \( N \)-point sequence is:

**Step 1.** Rearrange the original data points; their new position is found by reversing the bits of their indices.

**Step 2.** Repeat Step 3 \( \log N \) times. We will compute the DFT of the sequence starting with subsequences of size 1 and using Eq. (6.9) to build the DFT for all the larger-sized sequences up to size \( N \). Let \( S = 1 \) be the initial size of the sublists. \( S \) will double after each iteration.

**Step 3.** Repeat for all consecutive pairs of sublists in the array. There are \( N/(2 \times S) \) pairs.

Let the two sublists in this pair be \( G(k) \) and \( H(k) \) in Eq. (6.9). Compute \( X(k) \) for \( k = 0, \ldots, 2 \times S - 1 \). Store \( X(k) \) in the places previously occupied by \( G(k) \) and \( H(k) \).

Note that this algorithm can be used to compute both the DFT and the inverse DFT. Comparing Eq. (6.4) and (6.5) we see that we only have to change \( W_N^{kn} \) for \( W_N^{-kn} \) (\( W_N^{kn} \) and \( W_N^{-kn} \) are complex conjugates) and divide the final result by \( N \).

The Fast Fourier Transform can be readily implemented in parallel [9]. The equation for the DFT of size \( N \), Eq. (6.9), contains two subsequences of size \( N/2 \), which are DFTs that can be solved independently. This is the essence of the parallel algorithm. A separate process can be used to solve the DFT of each subsequence. To distribute the problem further, these subsequences can be divided in the same manner as the original DFT sequence with each subsequence being solved by a separate process. After the DFT has been computed for each subsequence, the two subsequences must be merged together as shown in Eq. (6.9). This merging process must be done for each process that is split.

The ideal number of processes (threads) should be \( N \), but we can spawn a smaller number depending on the available processors; it is easy to implement the tree structure described earlier: we create a *cleaner* thread that spawns two children and blocks until both children are done with their work; this implies that the parent *cleaner* will not be using any processing time while the children are working. When the worker
processes are all created, the tree structure is complete, with all parents waiting for their children to complete their work.

Each worker process needs a specific sublist of the original data points; no two workers share the same data. The parent cleaner will use this data after the children workers are done, and since the parents are blocked, there is no contention for write access to the array of data points. All cleaners and workers also require the values for the $W_N^k$ array, which is evaluated by the main thread and kept in shared memory where it is accessed by all the children threads.

6.2 Results

In this section, we describe the validation methodology and also present the results from simulations on the RPPT and actual executions of Presto programs on the Sun 3/60 and the BBN Butterfly.

In order to investigate the accuracy of the shared memory profiling technique and to estimate the overheads incurred, we compared the simulated execution times of selected Presto programs with the corresponding real execution times. We ran the Presto programs first on the Sun 3/60, since the shared memory profiler we used was designed for the MC68020 processor. We repeated this on the BBN Butterfly, because the Butterfly uses a different C compiler and therefore generates different assembly code. These experiments helped in the evaluation of the performance of the shared memory profiler.

We repeated the experiments on the Butterfly with the program data placed on remote nodes, so that all accesses to the data were forced to traverse the Butterfly network. From these runs, we obtained performance figures for the Butterfly architecture model, and how the error introduced by the architecture model affected the overall error of the simulation. In the following sections, we describe and analyze these experiments in detail.
For these quantitative studies, it was sufficient to consider sequential programs running on a uniprocessor because the shared profiling technique applies to a sequential section of code. The accuracy of simulating parallel programs depends on how accurately the profiler times the sequential sections of code within the processes that execute in parallel. The simulation of parallelism is accomplished by the discrete-event simulator that is a part of the RPPT; the profiler only generates the times when these events occur, by estimating the execution times of individual sequential sections of code.

6.2.1 Validation of the Shared Memory Profiler on the Sun

In order to examine the accuracy of shared memory profiling, we compared the actual running times (on the Sun 3/60) with the simulated times of two sequential algorithms - the Fast Fourier Transform (FFT) and the Subgraph Isomorphism algorithm (ISO).

For this validation, we first ran the Presto programs and then simulated their running on the RPPT with a dummy architecture model. In the second case, we used dummy routines for shared data accesses (which did nothing), since we were trying to study the accuracy of the profiling (of the instructions) rather than investigate the accuracy of the architecture model. The simulated times, therefore, reflect the times of execution of the program assuming no overhead due to shared variable accesses. This is what we want to see, since the Presto programs were run on the Sun, which does not differentiate between accesses. For the validation experiments on the Sun 3/60, it was imperative to use sequential programs, because a parallel Presto program runs on the Sun 3/60 as a pseudo-concurrent program (using multiprogramming), while a simulation on the RPPT would reflect parallel timings.
In the case of the Fast Fourier Transform, we changed the number of data points used from 1K to 32K. Figure 6.3 shows the execution and simulated times of the Fast Fourier Transform programs and the percentage differences.

<table>
<thead>
<tr>
<th>FFT # Points</th>
<th>Execution Time (secs)</th>
<th>Simulated Time (secs)</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>1.057</td>
<td>1.071</td>
<td>1.3</td>
</tr>
<tr>
<td>2048</td>
<td>2.260</td>
<td>2.323</td>
<td>2.8</td>
</tr>
<tr>
<td>4096</td>
<td>4.773</td>
<td>5.009</td>
<td>4.9</td>
</tr>
<tr>
<td>8192</td>
<td>10.887</td>
<td>10.745</td>
<td>-1.2</td>
</tr>
<tr>
<td>16384</td>
<td>22.493</td>
<td>22.940</td>
<td>2.0</td>
</tr>
<tr>
<td>32768</td>
<td>50.138</td>
<td>48.783</td>
<td>-2.7</td>
</tr>
</tbody>
</table>

Figure 6.3: Execution and Simulated Times - FFT

Figures 6.4 and 6.5 present the data of Figure 6.3 graphically. The program execution times varied from about 1 second to about 50 seconds. We see that the simulation shows a percentage error between -3 percent and 5 percent. This error is expected, since the timing is based on individual instruction timings, and does not take into account the combined effects of pipelining, caching, etc. There is no obvious pattern in the way that the error varies with the size of the program.

In the case of Subgraph Isomorphism, we used a larger graph that was fully connected and an arbitrary small graph. In such a case, each node in the smaller graph can map into any node of the larger graph, and the total number of isomorphisms is the total number of mapping permutations of the nodes of the smaller graph into the nodes of the larger. We varied the problem sizes so that the number of solutions varied from about 300 to about 12000.
Figure 6.4: Execution and Simulated Times - FFT

Figure 6.5: Percentage error in Simulation - FFT
Figure 6.6 shows the execution and simulated times and the percentage error.

<table>
<thead>
<tr>
<th># Isomorphisms</th>
<th>Execution Time (secs)</th>
<th>Simulated Time (secs)</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>336</td>
<td>0.304</td>
<td>0.311</td>
<td>2.3</td>
</tr>
<tr>
<td>360</td>
<td>0.471</td>
<td>0.499</td>
<td>5.9</td>
</tr>
<tr>
<td>720</td>
<td>0.626</td>
<td>0.678</td>
<td>8.3</td>
</tr>
<tr>
<td>1320</td>
<td>1.203</td>
<td>1.300</td>
<td>8.1</td>
</tr>
<tr>
<td>1680</td>
<td>1.956</td>
<td>2.069</td>
<td>5.7</td>
</tr>
<tr>
<td>5040</td>
<td>5.915</td>
<td>6.243</td>
<td>5.5</td>
</tr>
<tr>
<td>11880</td>
<td>14.268</td>
<td>15.359</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Figure 6.6: Execution and Simulated Times - ISO

Figure 6.7 and Figure 6.8 present the data of Figure 6.6 graphically. The program execution times varied from about 0.3 seconds to about 15 seconds. We see that the simulation shows a percentage error between 2 percent and 8.3 percent. This error, again, shows no obvious pattern that can indicate dependence of the error with data size. One possible reason for the variation of the error with the size of the data is the effect of the on-chip cache, which could alter the execution times depending on phases of cache locality.

These results show that the basic profiling step is quite accurate, and that we can generate accurate simulation timings by using such a technique. The addition of a model of the architecture to be simulated will introduce timing delays due to the actual remote references that are made by a program, but if we can represent these delays adequately, our technique for shared memory simulation using an execution-driven approach will be accurate.
Figure 6.7: Execution and Simulated Times - ISO

Figure 6.8: Percentage error in Simulation - ISO
6.2.2 Overhead of Profiling Presto Code

Execution-driven simulation of a program means that a program executes code until a "remote" operation is encountered. In our extension to the simulation of shared memory programs, we consider each access to a shared (remote) variable as a "remote" operation in the sense that event list manipulation is required and simulation-time is updated. Moreover, depending on the actual architecture being simulated, we may also need to do a detailed simulation of the way the remote access is handled.

Since we are running Presto programs, we need to also profile the Presto code in order to accurately represent the shared accesses made from within the Presto code. However, profiling the Presto code would imply that the processes would have to suspend (for timing update and simulation purposes) every time a shared variable was accessed from within the Presto code. Presto, however, has some critical sections of code (for shared memory synchronization), and in order to ensure that the program runs correctly and to completion, we cannot allow suspensions within these critical sections. For instance, if a process (thread) grabs a spinlock and suspends because of a shared access, then a thread arriving later spins on the same lock and the system deadlocks. This happens only in the simulation, since only one process can run at any given time (pseudo-concurrent) and only if certain critical sections of code are profiled.

In order to avoid this problem, we do not profile certain important sections of Presto code while creating the Presto library used in the simulation. In order to see how much this affects the simulation timings, we ran two sets of simulations of a number of Presto programs, one set with Presto code profiled and the other with none of the Presto code profiled. These programs were Subgraph Isomorphism (ISO), Gaussian Elimination (GAU), Fast Fourier Transform (FFT), Traveling Salesman (TSP) and the Game of Life (LIFE).
Figure 6.9 shows the execution time (simulated) of different application programs with the Presto code profiled and unprofiled. It also shows the number of data references added due to profiling the Presto code.

<table>
<thead>
<tr>
<th>Program</th>
<th>Execution Time (secs)</th>
<th># Remote References</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Presto Profiled</td>
<td>Presto Unprofiled</td>
</tr>
<tr>
<td>ISO</td>
<td>0.074</td>
<td>0.073</td>
</tr>
<tr>
<td>GAU</td>
<td>0.159</td>
<td>0.155</td>
</tr>
<tr>
<td>FFT</td>
<td>0.072</td>
<td>0.071</td>
</tr>
<tr>
<td>TSP</td>
<td>0.137</td>
<td>0.135</td>
</tr>
<tr>
<td>LIFE</td>
<td>0.212</td>
<td>0.208</td>
</tr>
</tbody>
</table>

Figure 6.9: Overhead of Profiling Presto Code

Figure 6.10 compares the simulation times of these programs, and Figure 6.11 compares the number of remote references made by the two sets of programs. Both these figures show that profiling all the Presto code does not affect the overall simulation timings significantly. Therefore, by not profiling certain small sections of Presto code, we are not any significant error in our simulation times.
Figure 6.10: Simulation Timings - Presto Profiling Overhead

Figure 6.11: Remote References - Presto Profiling Overhead
6.2.3 Overhead of Shared Memory Profiling

We have presented a technique for the simulation of parallel programs running on shared memory multiprocessors using execution-driven simulation. Our technique involves profiling the code to extract timing information at compile-time, and generating all the data addresses at runtime, in order to determine references to shared variables. For every data access, an overhead is incurred.

In order to calculate the overhead of our simulation technique, we use the following formula:

\[
\text{Overhead} = \frac{\text{Simulation Duration}}{\text{Simulated Time}}
\]

*Simulation Duration* is the real time taken for the simulation to run. *Simulated Time* is the program execution time as predicted by the simulator. To derive figures for the overhead of our execution-driven profiling technique rather than the overhead in the simulation of a particular architecture like the Butterfly, we used a special architecture model, where every data access was considered remote but did not result in any increase in the simulated time (although it did increase the simulation duration because of the event list manipulation and accounting). This approach represents the worst case scenario and reflects the efficacy of our technique.

Also, it was necessary to simulate sequential programs rather than parallel programs, because the simulation duration of a parallel program (for the same data size) would be larger while the simulated time would tend to decrease, reflecting speedup due to parallelism. This would lead to an overhead figure that would increase with the parallelism of the program.

Figure 6.12 shows the worst case simulation overheads of the Fast Fourier Transform algorithm for different data sizes and the corresponding execution times (simulated). The worst case assumption is that every data access is remote. We varied the data size from 256 points to 4K points. We see that the simulation overhead is about 360.
<table>
<thead>
<tr>
<th>FFT Points</th>
<th>Execution Time (secs)</th>
<th>Data References</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.244</td>
<td>60738</td>
<td>370</td>
</tr>
<tr>
<td>512</td>
<td>0.540</td>
<td>134211</td>
<td>357</td>
</tr>
<tr>
<td>1024</td>
<td>1.186</td>
<td>293596</td>
<td>358</td>
</tr>
<tr>
<td>2048</td>
<td>2.581</td>
<td>639045</td>
<td>360</td>
</tr>
<tr>
<td>4096</td>
<td>5.578</td>
<td>1380422</td>
<td>361</td>
</tr>
</tbody>
</table>

Figure 6.12: Worst Case Simulation Overhead - FFT

Figure 6.14 and Figure 6.15 depict the information of Figure 6.12 graphically. Figure 6.13 shows the worst case (architecture-independent) simulation overheads in the case of the Subgraph Isomorphism algorithm for a varying number of isomorphisms found, and the corresponding execution times (simulated). We varied the number of isomorphisms (solutions) from 120 to 11880. For these programs, we got overhead figures of around 250.

<table>
<thead>
<tr>
<th># Isomorphisms</th>
<th>Execution Time (secs)</th>
<th>Data References</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>0.119</td>
<td>12143</td>
<td>215</td>
</tr>
<tr>
<td>360</td>
<td>0.498</td>
<td>50474</td>
<td>217</td>
</tr>
<tr>
<td>720</td>
<td>0.676</td>
<td>76783</td>
<td>234</td>
</tr>
<tr>
<td>1320</td>
<td>1.296</td>
<td>153071</td>
<td>240</td>
</tr>
<tr>
<td>5040</td>
<td>6.242</td>
<td>725010</td>
<td>240</td>
</tr>
<tr>
<td>11880</td>
<td>15.357</td>
<td>1852946</td>
<td>252</td>
</tr>
</tbody>
</table>

Figure 6.13: Worst case simulation overhead - ISO

Figure 6.16 and Figure 6.17 depict the information of Figure 6.13 graphically.
Figure 6.14: Simulation Overhead - FFT

Figure 6.15: Simulated Execution Times - FFT
Figure 6.16: Simulation Overhead - ISO

Figure 6.17: Simulated Execution Times - ISO
The two algorithms used represent two different classes of programs. The Fast Fourier Transform algorithm involves floating point operations and also extensively uses mathematical library functions, while the Subgraph Isomorphism algorithm is non-numeric in nature. The reason for higher overhead figures in the Fast Fourier Transform case is that we did not profile some of the code from the mathematical libraries. We did get some of the source code for routines like \( \text{sin}() \), but this routine itself makes calls to other mathematical library routines which could not be profiled. Because these routines are called repeatedly, profiling the code would have increased the simulated execution times, giving lower overhead.

The overheads that we obtained are not unreasonable given that we extract every data address and use it for simulation. We could use a distribution as our simulation workload and be able to run faster simulations, but we would lose the accuracy and the program dependence that execution-driven simulation provides. Instruction-driven simulation, which could be slower than execution-driven simulation for any given section of code (due to the detailed emulation of each instruction), would incur additional overhead for any extra action based on every data address extracted. For such a detailed simulation, such overheads are acceptable.

Our approach has proven to be an effective, space efficient method that aids in the determination of performance based on a program making accesses to a shared data space which could be mapped to physical memory in any fashion.
6.2.4 Validation of the Shared Profiler on the Butterfly

The next step in our validation required running sequential Presto programs on the Butterfly and comparing the timings obtained on the Butterfly with the simulated values obtained from their RPPT simulations. The reason for this is that the Butterfly uses a different compiler and we wanted to examine the accuracy of the shared memory profiler on the Butterfly before investigating the overall accuracy of the Butterfly architecture model. This also enabled us to estimate how much of the error in our simulation was due to the profiler and how much was due to inadequacies in the architecture model of the Butterfly network. We used a 64-node Butterfly (GP1000) to run our programs. Before we could run any of the Presto programs on the Butterfly, it was necessary to port Presto onto the machine. This required a significant amount of effort, because the Butterfly C compiler uses registers for allocation of some local variables, unlike the Sun, which allocates local variables uniformly on the stack.

Also, since the Butterfly uses a different C compiler, we had to generate the assembly code on the Butterfly before we could profile it on the Sun 3/60 for simulation (steps 3, 4, 5 and 6 of the shared memory profiling - see Figure 4.1 in Chapter 4). This ensured that we were simulating exactly the same code as we executed on the Butterfly.

In order to validate the accuracy of the shared memory profiler, we ran three sequential algorithms on the Butterfly, using only a single processor for the program and having all the data located locally on the same processor. This prevents any effect of the Butterfly network on the execution times of the programs. The simulation times reflect only the effect of the profiling since all the data is local. There is no overhead due to remote accesses (the network does not play a role in the simulation timings).

The three sequential algorithms that we ran were the Fast Fourier Transform (FFT), Subgraph Isomorphism (ISO) and Gaussian Elimination (GAUSS).
Figure 6.18 shows the execution times (Butterfly) and the simulated times (Sun 3/60) of the Fast Fourier Transform algorithm on the Butterfly. All the data used by the program was locally available, and there were no requests across the network. We varied the number of data points from 1K to 32K. The execution times varied from about 1 second to about 50 seconds.

<table>
<thead>
<tr>
<th>FFT # Points</th>
<th>Execution Time (secs)</th>
<th>Simulated Time (secs)</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>1.118</td>
<td>1.154</td>
<td>3.2</td>
</tr>
<tr>
<td>2048</td>
<td>2.450</td>
<td>2.521</td>
<td>2.9</td>
</tr>
<tr>
<td>4096</td>
<td>5.320</td>
<td>5.469</td>
<td>2.6</td>
</tr>
<tr>
<td>8192</td>
<td>11.480</td>
<td>11.792</td>
<td>2.7</td>
</tr>
<tr>
<td>16384</td>
<td>24.850</td>
<td>25.293</td>
<td>2.6</td>
</tr>
<tr>
<td>32768</td>
<td>52.760</td>
<td>54.003</td>
<td>2.4</td>
</tr>
</tbody>
</table>

**Figure 6.18: Timing Information with Local Data - FFT**

Figure 6.19 compares the execution and simulation times. Figure 6.20 shows the percentage error of the simulation results from the actual Butterfly timings. The error is about 3 percent, showing that the shared memory profiling is very accurate in this case. Such an error is unavoidable due to the timing approximations made by the profiler, and the effects of pipelining and the on-chip instruction cache. Also, the error seems to be decreasing with the execution time of the program.
Figure 6.19: Execution (Butterfly) and Simulated (Sun) Times - FFT

Figure 6.20: Percentage Error in Simulation - FFT
Figure 6.21 shows the execution times (Butterfly) and the simulated times (Sun 3/60) of the Subgraph Isomorphism (ISO) algorithm on the Butterfly with only local data (no requests across the network) and the percentage error. We varied the number of isomorphism solutions from 2730 to 73440. The execution times varied from about 2 seconds to about 85 seconds.

<table>
<thead>
<tr>
<th># Isomorphisms</th>
<th>Execution Time (secs)</th>
<th>Simulated Time (secs)</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2730</td>
<td>2.230</td>
<td>2.060</td>
<td>-7.6</td>
</tr>
<tr>
<td>5040</td>
<td>4.941</td>
<td>4.438</td>
<td>-10.2</td>
</tr>
<tr>
<td>11880</td>
<td>11.888</td>
<td>10.887</td>
<td>-8.4</td>
</tr>
<tr>
<td>32760</td>
<td>34.761</td>
<td>32.579</td>
<td>-6.3</td>
</tr>
<tr>
<td>73440</td>
<td>83.505</td>
<td>79.720</td>
<td>-4.5</td>
</tr>
</tbody>
</table>

Figure 6.21: Timing Information with Local Data - ISO

Figure 6.22 compares the execution and simulation times. Figure 6.23 shows the percentage error of the simulation results from the actual Butterfly timings. The error in this case is negative and larger in magnitude, reaching a maximum of about 10 percent. In most cases, the error is about 6-8 percent. Again, as the program becomes larger, the error seems to decrease.
Figure 6.22: Execution (Butterfly) and Simulated (Sun) Times - ISO

Figure 6.23: Percentage Error in Simulation - ISO
The worst results were obtained in the third case (GAUSS). Figure 6.24 shows the execution times (Butterfly) and the simulated times (Sun 3/60) of the Gaussian Elimination program on the Butterfly with only local data (no requests across the network), and the percentage error of the simulation timings. We varied the matrix size from 50 to 175. The execution times varied from about 1.5 seconds to about 65 seconds.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Execution Time (secs)</th>
<th>Simulated Time (secs)</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1.490</td>
<td>1.741</td>
<td>16.8</td>
</tr>
<tr>
<td>75</td>
<td>5.051</td>
<td>5.826</td>
<td>15.4</td>
</tr>
<tr>
<td>100</td>
<td>12.012</td>
<td>13.753</td>
<td>14.5</td>
</tr>
<tr>
<td>125</td>
<td>23.551</td>
<td>26.794</td>
<td>13.8</td>
</tr>
<tr>
<td>150</td>
<td>40.709</td>
<td>43.223</td>
<td>13.5</td>
</tr>
<tr>
<td>175</td>
<td>64.689</td>
<td>73.314</td>
<td>13.3</td>
</tr>
</tbody>
</table>

Figure 6.24: Timing Information with Local Data - GAUSS

Figure 6.25 compares the execution and simulation times. Figure 6.26 shows the percentage error of the simulation results from the actual Butterfly timings. The error in this case is larger in magnitude, but again tends to be decreasing for the larger programs. It seems to be leveling off at about 13 percent. This error could be due to an inappropriate mix value used during profiling or inaccurate accounting for floating point operations.

From the results obtained from the sequential programs with local data, we see that the accuracy of the shared profiler varies across the applications, but gives fairly good estimates of the actual execution times of Butterfly code running on the MC68020 processor. The variation of the shared memory profiler behavior across applications is expected, because of the difference in the control flow of the programs. Different programs exhibit different patterns in the use of instructions, and therefore different usage of the processor’s on-chip cache and instruction pipeline.
Figure 6.25: Execution (Butterfly) and Simulated (Sun) Times - GAUSS

Figure 6.26: Percentage Error in Simulation - GAUSS
The instruction timings for the 68020 vary depending on pipelining and availability in the cache, and one set of timings cannot characterize any given application program's instruction mix entirely accurately. Each program is characterized by its own mix of instructions. Tracking the state of the instruction pipeline and on-chip cache would add a tremendous amount of overhead, and would require detailed instruction-driven simulation. Taking these factors into account, we can conclude from our results that the shared memory profiling technique is a viable technique for the estimation of execution times of code on a given processor.

6.2.5 Validation of the Butterfly Architecture Model

In order to validate the architecture model of the Butterfly (GP1000), we ran the same three sequential programs on the Butterfly. This time, we placed all the shared data used by the program on remote nodes, which forced the program to make remote accesses across the network. We compared the execution times (on the Butterfly) with the simulation times of the same programs (with identical remote data distribution in the simulation model) on the Sun 3/60. The simulation times incorporate the delays due to network latency, which are incurred by the program executing on the Butterfly. The reason why we ran the same sequential programs as we did in the validation of the shared profiler (only local data) was to be able to estimate how much of the error was introduced by the profiling step, and how much of the error was due to the inaccuracy in the Butterfly architecture model.

As described in the earlier section, we had to generate the assembly code on the Butterfly before we could profile it on the Sun 3/60 for simulation because the Butterfly uses a different C compiler. This ensured that we simulated exactly the same code as we executed on the Butterfly.

In Figure 6.27 we present the timing information derived from the Butterfly executions and from the simulations of the Fast Fourier Transform algorithm. All the shared data used by the program was available only on remote processor nodes. We
varied the number of data points from 1K to 32K. We noticed that placing the data remotely almost doubled the execution times of the programs on the Butterfly in this case. The execution times varied from about 2 seconds to about 90 seconds. For this case (FFT), the average overhead for the simulations was about 780. The remote references constituted almost 50 percent of the total data references, implying detailed simulation of packets across the Butterfly network for one of every two accesses. Comparison with the Subgraph Isomorphism case (next case) shows an increase in the overhead from about 150 to about 780 for an increase in the percentage of remote references by a factor of 10. This is less than an order of magnitude increase in the overhead, despite detailed simulation of the Butterfly network.

<table>
<thead>
<tr>
<th>FFT # Points</th>
<th>Execution Time (secs)</th>
<th>Simulated Time (secs)</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>1.873</td>
<td>1.826</td>
<td>-2.5</td>
</tr>
<tr>
<td>2048</td>
<td>4.096</td>
<td>3.991</td>
<td>-2.6</td>
</tr>
<tr>
<td>4096</td>
<td>8.906</td>
<td>8.656</td>
<td>-2.8</td>
</tr>
<tr>
<td>8192</td>
<td>19.165</td>
<td>18.662</td>
<td>-2.6</td>
</tr>
<tr>
<td>16384</td>
<td>41.250</td>
<td>40.023</td>
<td>-3.0</td>
</tr>
<tr>
<td>32768</td>
<td>88.198</td>
<td>85.445</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

Figure 6.27: Timing Information with Remote Data - FFT

Figure 6.28 compares the execution times (Butterfly) and the simulated times (Sun 3/60) of the Fast Fourier Transform algorithm on the Butterfly. Figure 6.29 shows the percentage error of the simulation results from the actual Butterfly timings. The error is about -2.5 percent, showing that the overall simulation is very accurate in this case. Also, the trend in the seems to be the same as in the case with local data.
Figure 6.28: Execution (Butterfly) and Simulated (Sun) Times - FFT

Figure 6.29: Percentage Error in Simulation - FFT
Figure 6.30 shows the timing information derived from running (on the Butterfly) and simulating (Sun) the Subgraph Isomorphism algorithm for different sizes of input graphs, with shared data placed on remote nodes. We varied the number of isomorphism solutions from 2730 to 73440. In this case, the execution times on the Butterfly did not change significantly from the case with all data local. The reason for this was that only about 5 percent of the data references were to shared data across the network. The execution times varied from about 2 seconds to about 90 seconds.

<table>
<thead>
<tr>
<th># Isomorphisms</th>
<th>Execution Time (secs)</th>
<th>Simulated Time (secs)</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2730</td>
<td>2.405</td>
<td>2.130</td>
<td>-11.4</td>
</tr>
<tr>
<td>5040</td>
<td>5.397</td>
<td>4.751</td>
<td>-12.0</td>
</tr>
<tr>
<td>11880</td>
<td>12.334</td>
<td>11.568</td>
<td>-10.6</td>
</tr>
<tr>
<td>32760</td>
<td>37.245</td>
<td>34.329</td>
<td>-7.8</td>
</tr>
<tr>
<td>73440</td>
<td>88.742</td>
<td>83.447</td>
<td>-6.0</td>
</tr>
</tbody>
</table>

Figure 6.30: Timing Information with Remote Data - ISO

Figure 6.31 compares the execution times (Butterfly) and the simulated times (Sun 3/60) of the Subgraph Isomorphism (ISO) algorithm on the Butterfly with all data remotely located (remote references across the network). Figure 6.32 shows the percentage error of the simulation results from the actual Butterfly timings. The error in this case is larger, reaching a maximum of about -12 percent, going down to about -6 percent for the largest test case simulated. The trend in the error is the same as with the simulations with no remote data. The overheads for simulation in this case are much lower, going down to about 150.
Figure 6.31: Execution (Butterfly) and Simulated (Sun) Times - ISO

Figure 6.32: Percentage Error in Simulation - ISO
Figure 6.33 shows the timing information derived from the Gaussian elimination algorithm with the matrix data situated on a node remote from the node operating on it. We varied the matrix size from 50 to 175. In this case, the execution times on the Butterfly increased by a factor of 3, since 62 percent of the data references were remote. The execution times varied from about 4 seconds to about 175 seconds.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Execution Time (secs)</th>
<th>Simulated Time (secs)</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>3.928</td>
<td>4.293</td>
<td>9.3</td>
</tr>
<tr>
<td>75</td>
<td>13.387</td>
<td>14.514</td>
<td>8.4</td>
</tr>
<tr>
<td>100</td>
<td>32.047</td>
<td>34.438</td>
<td>7.5</td>
</tr>
<tr>
<td>125</td>
<td>62.914</td>
<td>67.301</td>
<td>7.0</td>
</tr>
<tr>
<td>150</td>
<td>108.93</td>
<td>116.342</td>
<td>6.8</td>
</tr>
<tr>
<td>175</td>
<td>173.095</td>
<td>184.799</td>
<td>6.8</td>
</tr>
</tbody>
</table>

Figure 6.33: Timing Information with Remote Data - GAUSS

Figure 6.34 shows the execution times (Butterfly) and the simulated times (Sun 3/60) of the Gaussian Elimination program on the Butterfly with the Matrix (shared data) located on a remote node, forcing remote accesses across the network. Figure 6.35 shows the percentage error of the simulation results from the actual Butterfly timings. The error in the simulation starts at about 10 percent and decreases until it levels off at about 7 percent, showing relatively accurate simulation. The simulation overheads were the highest in this case, about 1200. This high value is because of the high percentage of remote references - 62 percent.
Figure 6.34: Execution (Butterfly) and Simulated (Sun) Times - GAUSS

Figure 6.35: Percentage Error in Simulation - GAUSS
6.3 Overview of results

The Butterfly architecture model is sufficiently accurate in all the cases and is always negative. In all cases, simulation of the Butterfly network had the effect of adding a negative error to the case without the network. If the simulation timings of the shared profiler were precise, the Butterfly architecture model would introduce a maximum negative error of about 5 percent, which is indicated by both the cases (FFT and GAUSS) which have a significant number of remote accesses and where the Butterfly architecture model contributes significantly to the overall simulated time. In the case of Subgraph Isomorphism, the percentage of remote references is only about 5 percent, and we do not see much change in the percentage error from the case with only local data.

A part of this negative error (simulation time less than execution time) is likely due to the fact that our simulations do not take into consideration any network traffic due to other application programs running on other nodes. The execution times from the Butterfly were obtained when other users were also on the system, and there was network traffic originating from many other nodes. We could be introducing a positive error by modeling the Butterfly switching network by a single process, thereby making it appear synchronous.

The error due to shared profiling clearly changes with the program. This is because of different patterns of instruction access by the programs. It is not easy to find a mix for the timings that is applicable across a range of programs, and this is a limitation of the profiling technique. Instruction execution times change with pipelining, caching, etc., and this could happen even across runs of a single application with different data sizes.

The combined effects of shared profiling and the Butterfly architecture model always simulated the programs we tested with an error of less than 10 percent. Thus, we can conclude, from the available data, that our simulation technique is reasonably accurate. In two of the cases (FFT and GAUSS), the errors due to profiling and the
errors due to the architecture model offset each other. This is not true in the third case (ISO). Thus, we cannot guarantee that the profiler error and the architecture model error will always offset each other. From the values of the simulation overhead, we see that the overhead increases with the percentage of remote references. Each remote reference introduces overhead due to network access that needs to be simulated in detail. Since the overhead is as low as 150 in the Subgraph Isomorphism case, we can conclude that most of the overhead is introduced by the network model, and not by the profiler. Therefore, shared profiling is a viable simulation technique for the simulation of shared memory multiprocessors, and the overall simulation overheads incurred by this technique will depend mainly on the complexity of the architecture being simulated (the complexity of the architecture model).
Chapter 7

Evaluation

In this section, we present an evaluation of our technique for the simulation of shared memory parallel systems. We restrict ourselves to the evaluation of the shared memory profiling technique.

7.1 Implications of the Results

The results that were presented in Chapter 6 give an indication of the overheads involved and the accuracy that can be obtained from shared memory profiling. There are two sets of results that indicate the accuracy of the shared profiling technique. The first set shows the profiler validation on the Sun 3/60 with a dummy architecture model (Figures 6.3 through 6.8). The second is the set of simulation timings with the Butterfly architecture, but with no remote data (the Butterfly architecture model does not play a role in the timings - Figures 6.18 through 6.26). Both sets of data indicate in most cases that the shared profiling technique is accurate to within about ±8 percent.

As we saw in Figures 6.12 and 6.13, the worst case overhead of the simulation technique varies between 250 and 350 for the two test cases. The worst case assumption is that every data reference is to shared memory and requires event list manipulation. This is not true, as we saw in the simulations with shared data, where the percentage of remote references varied from 5 percent (ISO) to 62 percent (GAUSS). In fact, the minimum overhead in the case of ISO was 150, which is quite low, considering
that each data reference is examined. We can conclude from these results that our shared memory profiling technique is sufficiently accurate to be useful and sufficiently efficient to be practical.

7.2 Limitations

In the current implementation, we insert timing information at the beginning of each basic block. Each basic block may contain a number of remote data accesses. When we encounter the first remote access, we account for the time for the entire basic block, rather than just for those instructions before the data access. The accuracy of the simulation could be enhanced by inserting timing information for the code between every two remote accesses rather than at the beginning of each basic block. This would make the simulation more accurate but would increase the simulation overhead.

The timing analysis performed by the profiler is not entirely accurate. Since timing analysis must be performed at compile-time, instructions such as shifts and branches may introduce inaccuracies into the analysis. This is because the time required by these instructions at run-time depends on the values of the operands, quantities that are usually not available during the compile-time analysis of the program. In order to overcome this problem, an increased overhead is required in determining the values of the data dynamically.

Currently, timing analysis is implemented as a table-driven lookup of timing costs for each instruction, which requires a scan and parse of instructions to break them up into opcode and operands. Timing information is supplied by the manufacturer's documentation for the processor. The 68020 is a pipelined processor with an on-chip instruction cache. The documentation for the 68020 attempts to capture pipeline and cache effects by quoting several timing costs for each instruction, one cost for each of several somewhat idealized execution conditions. Each 68020 instruction is assigned a best-case, cache-case, and worst-case execution cycle count. The best-case cost reflects
the time for instruction execution when the instruction is in the on-chip cache and benefits from maximum overlap due to other instructions. The cache-case reflects the time when the instruction is in the cache but has no overlap. The worst-case reflects the time when the instruction is not in the cache, or the cache has been disabled, and there is no instruction overlap. For a given program, the timing analysis must choose which weighted average of cases best represents the typical program execution conditions. While the on-chip cache can be turned off for validation purposes, the pipelining effects have not been accounted for in the current implementation, and add to the error in predictions by the profiler.

A serious limitation of the shared memory profiling technique is that it does not support cross-profiling [10, 12] directly. Cross-profiling is used when the host and target processors differ. It attempts to match the basic blocks of the program code that is compiled on the host and target machines. The timing information introduced into the host code is changed to reflect those of the corresponding (matching) basic blocks of the target code. The shared memory profiler also generates data addresses at run time, and these are likely to differ on the host and the target processors. There is no direct way in which these addresses can be matched from the target code to the host code, although the timing information can be extracted from the target code. Since we are extracting information from the data addresses about the nature of the corresponding references, rather than use the actual address value itself, this problem can be alleviated by a proper mapping of the address space of the target architecture to that of the simulation host. This approach would, however, be inaccurate if the number of data references made by the host and target code differs significantly, as could happen with two processors with very different compilers or widely varying instruction sets.

Another limitation, at least with respect to uniformity, is that sections of code that deal with synchronization explicitly cannot be profiled. Although this may not
cause any significant error in the simulation timings, as mentioned in Section 6.2.2, it could lead to deadlock in the simulation unless care is taken to avoid deadlock.

7.3 Possible Enhancements

Currently, the distribution of data (allocation of data to processors) is performed by a user-provided routine, and there is no automatic way of specifying data distribution, even if it is uniform. The user must specify the ranges of addresses associated with every node, and there is currently no default setting. Routines supporting data distribution would improve the programming system being used for simulation.

The design of the shared memory profiler was patterned after the design of the cache profiler described by Dwarkadas [17], which has been designed for the simulation of systems with caches. This was done to facilitate the integration of these two techniques, so as to be able to simulate large scale shared memory multiprocessor systems with external data caches. Our shared memory profiler currently does not address the simulation of caches, but this extension is both feasible and desirable.

There are some possibilities of adding low-overhead mechanisms to the profiler to include the effects of pipeline stalls on branches, and these could improve the accuracy of the profiler timing estimates.

The applicability of cross-profiling to the shared memory profiling technique could also be examined, and modifications to the shared memory profiling technique based on cross-profiling is an area for future research. This would involve studies of the instruction sets of the host and target processors and how their reference patterns differ. It would also involve studies of the compilers used on the two machines.
Chapter 8

Conclusions

In this section we present the conclusions of the work presented in this thesis. Directions for future work in the simulation of shared memory systems are also pointed out.

8.1 Conclusions

Accurate simulation of computer systems is essential in the analysis and performance evaluation of computer systems, in the design of new systems, and in the matching of algorithms to architectures. It is possible to generate this information by representing a computer program as a statistical workload, but the results generated are usually not sufficiently accurate. The trace-driven approach is often inappropriate in the simulation of parallel computer systems. Emulation of the instructions of a program provides accuracy but incurs significant overheads. We have investigated the feasibility of using execution-driven simulation, where the simulation of an architecture is driven by the execution of the program itself, for simulating shared memory parallel systems.

We have designed and validated extensions to this technique to support the simulation of shared memory multiprocessors. These extensions were demonstrated by simulating actual shared memory programs, written using Presto, on the Rice Parallel Processing Testbed.

We have described a shared memory profiler, which profiles the assembly code generated by an application program. The profiler inserts timing information for each
basic block, and also inserts code at compile-time that generates the data addresses accessed by the program at run-time. The timing information is used to update the simulation time during execution, while the data addresses are examined in order to determine the nature of the reference. The data references are simulated depending on the architecture being simulated.

For validation, we ran three application programs on the BBN Butterfly (GP1000), a shared memory multiprocessor. The Butterfly has 64 MC68020 processor nodes connected by a Butterfly switching network. We developed an architecture model for the Butterfly to support our simulation. We validated the shared memory profiler on the Sun 3/60 as well as on a single node of the Butterfly, and found that the shared memory profiling technique gives results that were accurate to within 8 percent to 10 percent in most cases. The overhead of the simulation technique was found to be between 250 and 350 in the worst case (assuming every reference is a remote reference and incurs events list manipulation activity). This overhead is acceptable, considering that every data access made by the program is examined.

We also obtained timing information by running programs on the Butterfly with the data located remotely. These timings include overheads due to network accesses. Simulation of these programs with identical distributions of the data gave us execution times very close to those obtained on the Butterfly, showing that the Butterfly architecture model that we developed was quite accurate. The percentage error due to the Butterfly architecture alone was always negative and was always within 5 percent. For these simulations, there was an additional overhead due to the detailed simulation of the remote accesses across the Butterfly network. The overall overhead ranged from about 150 to about 1200, and this depended directly on the percentage of remote references in the program, which ranged from about 5 percent to 62 percent. Based on these results, we conclude that execution-driven simulation of programs on shared memory multiprocessors is both accurate and viable.
8.2 Future Work

The decision to use a profiling technique that extracts data addresses at run time is a direct outcome of the design decision to support extension of the shared memory simulation technique described in this thesis to include local caches. It is difficult to envisage large, shared memory multiprocessors of the future with no high speed local caches.

Such cache organizations would include some mechanism to maintain cache coherency, and detailed simulation of caches in a shared memory multiprocessor environment would entail extraction of instruction and data addresses. In order to extend our model to include the simulation of caches in shared memory multiprocessors, we need to extract the instruction addresses and simulate the cache behavior along with the shared reference behavior that is currently being simulated. Work has been done in the execution-driven simulation of caches [17]. Instruction addresses can be extracted at compile-time, thus reducing simulation overheads.

Another possible area for future research is the parallelization of the simulation. The RPPT is a sequential discrete-event simulator, and any sequential simulator is limited by the size of simulation that it can practically handle, due to constraints of space, time and processing power. Parallel simulation is an attractive alternative, especially when the overhead of simulation is not small.

Parallelization of the simulation would make simulation of larger programs tractable, thus allowing us to obtain important data that could be used in the design of large-scale shared memory multiprocessors. Parallel simulation is itself a difficult research problem. There is some ongoing research to implement execution-driven simulation on the Hypercube. Some work has been done on execution-driven parallel simulation on a shared memory multiprocessor [25], but much work remains.
Bibliography


